GLENTOP

DR WATSON series

EINSTEIN ASSEMBLY LANGUAGE COURSE

by
Mike Bayliss
with software by
Graham Rounce

EINSTEIN ASSEMBLY LANGUAGE COURSE

EINSTEIN

ASSEMBLY LANGUAGE COURSE

by

Dr M. D. Bayliss

software by

Graham Rounce

GLENTOP

PUBLISHERS □ LIMITED

FEBRUARY 1986

All programs in this book have been written expressly to illustrate specific teaching points. They are not warranted as being suitable for any particular application. Every care has been taken in the writing and presentation of this book but no responsibility is assumed by the author or publishers for any errors or omissions contained herein.

COPYRIGHT © Glentop Publishers Ltd 1986
World rights reserved.

No part of this publication may be copied, transmitted or stored in a retrieval system or reproduced in any way including but not limited to photography, photocopy, magnetic or other recording means, without prior permission from the publishers, with the exception of material entered and executed on a computer system for the reader's own use.

ISBN 1 85181 010 2

Published by:

Glentop Publishers Ltd
Standfast House
Bath Place
High Street
Barnet
Herts EN5 5XE
Tel: 01–441–4130

Disc formatted with DOS 1.31 used under licence
from Crystal Research, being part of
ISBN 0-9506-828-3-7

Printed in Great Britain by
Antony Rowe Ltd., Chippenham

Contents

Introduction

Chapter 1

Getting Started • Z80 Chip Architecture • Immediate Addressing • Register – Register Addressing • RST 08 & DEFB 158 • Summary

Chapter 2

Jumping, Subroutines & Labels • Unconditional Jumps • The Program Counter • Conditional Jumps • Labels • Summary of Labels • The Flags • Summary

Chapter 3

Register Pairs & Addressing Modes • Indirect Addressing • Indexed Addressing • Summary

Chapter 4

Arithmetic Operations • Subtraction • The Carry Flag Instructions

Chapter 5

Binary Coded Decimal & Logical Operators • Signed Numbers • One's Complement • Two's Complement • Summary

Chapter 6

Multiplication, Division & The Rotate Group • Binary Multiplication • 8–bit Binary Multiplication • The Bit Set, Reset & Test Group • Summary

Chapter 7

The Stack • PUSH & POP • The Stack Pointer • Summary

Chapter 8

Block Moves • Parity • Compares • Summary

Chapter 9

Special Operations & Interrupts • Interrupts • Interrupt Modes • Alternate Registers • More EXchange Instructions • The INs & OUTs of the Z80 • NOP • Summary

Chapter 10

Einstein Graphics • Graphics Functions • Fill Routines • Sprite Handling in Assembly Language • Sprite Shapes

Appendix 1

The Z80 Instruction Set

Appendix 2

Effects Of Instructions on the Flags • The Flags

Appendix 3

The Effects Of Comparisons on the Overflow, Sign & Carry Flags

Appendix 4

Some Built–in Routines

Appendix 5

Binary, Binary–Coded Decimal & Hexadecimal Notation

Appendix 6

The Assembler • Summary of Commands

Appendix 7

ASCII Codes

Glossary

Solutions

Index

Introduction

This book forms part of the Dr Watson Assembly Language Series. It has been written so as to provide a completely self–paced course in Z80 Assembly Language Programming. Throughout the course, new instructions are illustrated with example programs. Also, numerous exercises have been included to help test the understanding of the concepts involved.

After reading and working through the first couple of chapters, the reader might find it helpful to read Appendix 6. This Appendix summarises the use of the assembler, including how to save and load programs on disc.

Whilst working your way through the book don't be afraid to experiment and try out any ideas you have, you will not harm the computer or Assembler in any way.

Well that's the spell finished: you now have ten action–packed chapters, not to mention the appendices, to work through! I hope you enjoy it.

Dr M. D. Bayliss

Chapter 1

Getting Started

Having used a computer before, you are probably well aware that things called 'machine code' and 'assembly language' exist. Quite simply, machine code is the language that the microprocessor chip, a Z80, in your computer understands. As an example, take a simple addition sum – adding one to 83.

In English you would say:

Add one to eighty three; what's the answer?

In BASIC you might say something like:

```
1Ø LET A=83
2Ø LET A=A+1
3Ø PRINT A
```

In Z80 machine code you could say:

```
3E 53
C6 1
CF 9E
C9
```

This is pretty--well unintelligible, isn't it? Well, that's why we use assembly language. The same program is given below in assembly language along with a brief comment on each line.

```
LD A,83          Load the Accumulator 'A' with '83'
ADD A,1          Add 1 to the Accumulator
RST 8            }
DEFB 158         } Store the contents of A on the screen
RET              Return from the machine–code subroutine
```

That's much easier to read than the machine code, isn't it? With an assembler you can enter your program in assembly language and be able to read though it and understand it more readily. All the assembler does is to change the assembly language into machine code. Thus, when it sees 'LD A' it changes this command into the machine code '3E' and puts this into memory in the right place.

The Z80 chip contains various registers, in effect internal memory locations, where data can be stored. Most of the Z80's instructions use these registers. One of the most frequently used registers is the accumulator. The accumulator is capable of storing numbers from 0 to 255. Let's now look at an instruction which allows us to LoaD a value into the accumulator. The instruction for this is:–

LD A,n	**LoaD** the **A**ccumulator with the immediate value **n**

You will notice that the letters chosen for the instruction reflect what it accomplishes. Instructions of this form are called 'mnemonics'.

There are many different ways of getting numbers into and out of the accumulator and other parts of the computer; these different ways are known as 'addressing modes'. The method above, which uses an unmodified typed–in number, is known as 'immediate–mode' addressing.

The next instruction in the program is ADD:

ADD A,n	**ADD** the immediate value **n** to the **A**ccumulator

This instruction takes the number following the comma, in the example '1' (on page 1–1), and adds it to the value currently held in the accumulator (A). The answer is left in the accumulator, deleting the old number, in our case 83.

Looking back at the example, probably the most baffling instructions are:–

```
RST 8
DEFB 158
```

These tell the computer to store the contents of the accumulator, 84, on the screen – i.e. to display the contents. Whilst true, this is really only half the story.

The computer has many built–in machine–code programs or 'routines' stored in its chips. Without these, it would not be able to understand BASIC, for instance. One of these routines is designed to take a value from the accumulator and display it on a monitor.

To get at these routines in order to use them, RST 08 is used. The RST refers to RESTART, and directs the computer to a special inbuilt function. Which function the computer performs will be decided by the DEFined Byte which follows the ReSTart. Inside the computer, these functions are stored in specific memory locations; a full list of these Machine CALls (MCALS) is given in Appendix 4.

Right then! Let's have a go at running a machine–code program!

A couple of points about the assembler: when you start to write a program the assembler should be told where you want the program to be placed in the computer's memory, first of all when it is being typed in and assembled, and secondly, when it is run. Generally, a good place for machine–code programs to start from is memory location 256 (the computer uses locations 0 to 255). However, the assembler uses locations 256 to 11007 inclusive. Thus, when we are actually typing in and assembling programs, they will have to go elsewhere. For reasons which will become apparent later, a good place for the code to start is memory location 20224. To get the assembler to assemble to this location, the direct–entry command 'O' is used, as will be shown below. When the machine–code program is actually run, we will

make it start at location 256. The assembler directive to do this is ORG. To make life easy for itself, the computer adds the O and ORG values together when assembling the program so the assembled code in fact goes at 20480. (Note: a 'direct–entry command' goes outside a program, a 'directive' goes within a program. Both, however are instructions to the assembler and do not get translated into machine code.)

Thus the first line of the program will be:

```
1Ø ORG 256
```

.

.

.

program

.

.

.

You must tell the computer to return from machine code to BASIC (or the operating system). The command that does this is RET.

RET	**RET**urn from machine–code subroutine

Right, to make the example into a program, we must:

1. Tell the assembler to decide where to store the assembled program (i.e. from memory location 20224+256).

2. Tell the assembler to which memory location it should load and run the program.

3. LoaD immediate value '83' into the Accumulator,i.e. 'LD A,83'

4. Add 1 to the contents of the accumulator. The mnemonic for this is 'ADD A,1'

5. Use the built–in machine–code routine to display the new value in the accumulator on the screen, i.e. 'RST 08' and 'DEFB 158'

6. RETurn from our machine–code program to the operating system, i.e. 'RET'

7. Tell the assembler (not the Z80) that the program has finished, using an '@'.

or

PROGRAM 1.1 (don't type this in just yet)

```
1Ø ORG 256
2Ø LD A,83
3Ø ADD A,1
4Ø RST 8
5Ø DEFB 158
6Ø RET
```

Now to enter this:

a) Load the ASSEMBLER program into the computer as follows: place the assembler disc in drive 0 with side A uppermost and switch on. After the '0:' prompt appears, type 'ASSEM' and then press ENTER. The assembler will load into the Einstein's memory.

b) If the ALPHA LOCK light is out, press ALPHA LOCK so you're typing in capital letters, then type 'O&4F00' without the quotes, of course (i.e. type the letter O, SHIFT–6, 4, F, F, zero, zero). The &4F00 is computer–code for 20224, and will be explained in Chapter 4. Then press ENTER.

c) Type 'A', to begin entering a new program, then press ENTER.

d) A 10 will appear; type in 'ORG 256' (don't leave out the space) and then press the ENTER key.

e) A new line number will now appear, 20. Type in the next line of the program, 'LD A,83' and press the ENTER key as before. Don't leave out the space between the 'LD' and the 'A'. Also, do not insert a space between the comma and the number, or indeed anywhere else.

f) 30 will now appear. Type in 'ADD A,1', then the ENTER key. Again, don't leave out the space between ADD and A.

g) On line 40, type in 'RST 08' (that's zero eight) and then press the ENTER key. There should be a space between RST and 08.

h) On line 50, type in 'DEFB 158'. Then press the ENTER key.

i) On line 60 simply type RET and press the ENTER key.

j) Now, as we have finished typing in the program, and want to return to the command mode, press @ (i.e. hold down SHIFT and press the zero key) and then the ENTER key.

k) Now press V then ENTER to view the program. The program should now appear as below:

```
1Ø ORG 256
2Ø LD A,83
3Ø ADD A,1
4Ø RST 8
5Ø DEFB 158
6Ø RET
```

If for any reason it doesn't then note the number of the line which is wrong and re–enter the correct data. You edit a line simply by typing in the correct line, including the line

number and then pressing ENTER. This is not the case so press @ then ENTER. When your program appears as above, when listed, press Z (and ENTER, as usual). The assembler will now assemble the program into machine code.

l) After the assembler has assembled the program, if there are any errors in the program, the assembler will inform you. Re–type and re–assemble the offending lines, as described in (k) above.

m) To save your program on disc, use the **W**rite file option (i.e. press W). Save the Assembled code using any file name of your choice, e.g. 'W PROG' followed by ENTER and then **Q**uit the assembler ('Q' ENTER). Now simply run your program from DOS (Disc Operating System) by typing in its name (e.g. 'PROG') followed by ENTER. If the message 'No file' appears then either you have typed the name incorrectly, or you have used a file name which is not allowed (e.g. 'P1__1'); 'FILE NOT FOUND' probably means you are using the wrong disc.

In case you're wondering why the program puts a capital T on the screen and not the answer, 84, have a look at Appendix 7 which shows the Einstein's version of the 'ASCII character set'. You will see that the code for 'T' is 84.

ASCII stands for the 'American Standard Code for Information Interchange'. Every text symbol which the computer can put on the screen has its own ASCII code number. Thus when asked to put 84 onto the screen, the routine called using RST 08 and DEFB 158 puts character number 84 onto the screen.

This is true in BASIC as well, even when doing a sum like 'PRINT 83+1'. In this case, although the computer puts '84' on the screen, it has had to find out what the characters '83+1' mean, do the appropriate sum and convert the answer using the ASCII codes for the characters '8' and '4'. This is in fact quite complicated to do. The computer has a built–in routine which does this automatically for BASIC.

number and then pressing ENTER. This is not the case so press @ then ENTER. When your program appears as above, when listed, press Z (and ENTER, as usual). The assembler will now assemble the program into machine code.

l) After the assembler has assembled the program, if there are any errors in the program, the assembler will inform you. Re–type and re–assemble the offending lines, as described in (k) above.

m) To save your program on disc, use the **W**rite file option (i.e. press W). Save the Assembled code using any file name of your choice, e.g. 'W PROG' followed by ENTER and then **Q**uit the assembler ('Q' ENTER). Now simply run your program from DOS (Disc Operating System) by typing in its name (e.g. 'PROG') followed by ENTER. If the message 'No file' appears then either you have typed the name incorrectly, or you have used a file name which is not allowed (e.g. 'P1__1'); 'FILE NOT FOUND' probably means you are using the wrong disc.

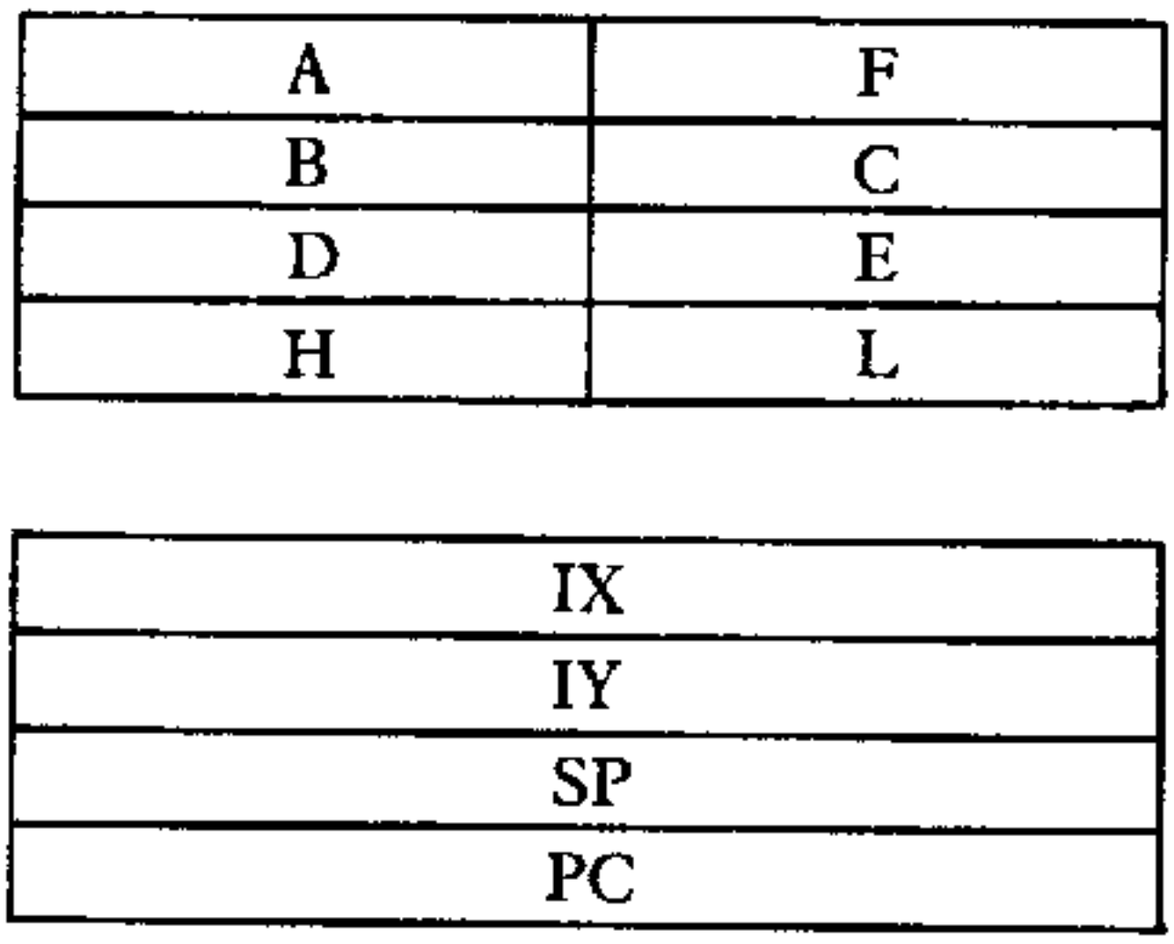
In case you're wondering why the program puts a capital T on the screen and not the answer, 84, have a look at Appendix 7 which shows the Einstein's version of the 'ASCII character set'. You will see that the code for 'T' is 84.

ASCII stands for the 'American Standard Code for Information Interchange'. Every text symbol which the computer can put on the screen has its own ASCII code number. Thus when asked to put 84 onto the screen, the routine called using RST 08 and DEFB 158 puts character number 84 onto the screen.

This is true in BASIC as well, even when doing a sum like 'PRINT 83+1'. In this case, although the computer puts '84' on the screen, it has had to find out what the characters '83+1' mean, do the appropriate sum and convert the answer using the ASCII codes for the characters '8' and '4'. This is in fact quite complicated to do. The computer has a built–in routine which does this automatically for BASIC.

Z80 Chip Architecture

The Z80 has various registers other than the accumulator, and a simplified diagram of the arrangement is shown below, in Figure 1.1.



Simplified Z80 Architecture

FIGURE 1.1

The accumulator, or A register, you have already met. It is somewhat special in that there are more Z80 instructions which can operate on it than there are for the other registers.

All of the registers (just memory locations within the Z80 chip) are 'eight–bit registers', that is, they can only store numbers from 0 to 255. This may not seem terribly useful, but there are ways of using the registers in combination to store much bigger numbers, as will be seen later.

The H and L registers taken together are known as the 'primary data pointer'. Registers B and C, and D and E when taken as pairs, are called 'secondary data pointers'. The main point about this for now though is that instructions which use H and L are often shorter when assembled into machine code, and therefore run faster.

Immediate Addressing of the Registers

Immediate addressing of the accumulator was discussed earlier, where a number was loaded into the accumulator using the instruction

```
LD A,83
```

Exactly the same form of instruction can be used to load numbers from 0 to 255 into any of the other registers, e.g.

```
LD B,83
```

would load the number 83 into the B register.

In general,

LD r,n	**LoaD** register **r** with the immediate value **n**

The 'r' in the above box stands for any register, i.e. any one of A, B, C, D, E, H, or L.

Register – Register Addressing

Not only can any register be loaded with a number, but any register can be copied into any other register, or into the accumulator. For example, Program 1.2 puts 43 into the L register, copies this into the accumulator, and displays it on the screen – an ASCII '+'.

PROGRAM 1.2

```
1Ø ORG 256
2Ø LD L,43      Immediate-mode load of L with 43.
3Ø LD A,L       Load A with contents of L.
4Ø RST 8        Display the contents of the
5Ø DEFB 158     accumulator.
6Ø RET          Return to operating system.
```

Line 30 contains the new instruction:

```
LD A,L
```

This copies the contents of register L into the accumulator,

Another example could be:

```
LD B,E
```

or

```
LD C,A
```

In general,

LD r1,r2	copy the contents of register **r2** into register **r1**

Some authors refer to register–register addressing as 'inherent' addressing, and others refer to it as 'implied' addressing. If you refer to other books, beware of their terminology, as this can lead to confusion.

Right then! Let's type in Program 1.2. If you make a mistake typing in any line before pressing the ENTER key, just use the DEL key to rub out the mistake and type in the correct version. If you notice a mistake on a line after you've pressed the ENTER key, you can replace those lines after pressing @ and before assembling the program. You could replace them after assembling the program, but don't forget to re–assemble it afterwards.

1. Run the assembler program unless it is loaded already.

2. Enter 'O&4F00' to say where the assembled code is to be stored.

3. Enter 'A' to begin entering the program, and make sure it is in capital letters mode.

4. Tell the assembler where in memory the program is to be run, i.e. type in 'ORG 256'.

5. Type in 'LD L,43' and press the ENTER key, (press the ENTER key after each entry).

6. Type in 'LD A,L'

7. Type in 'RST 08'.

8. Type in 'DEFB 158'

9. Type in 'RET'

10. Press @

11. Use 'V' to view the program: check it!

12. Enter 'Z' to assemble the program.

13. Enter 'W' and a filename to write the file to disc.

14. Enter 'Q' to quit the assembler.

15. Type your program's name to run it.

After that, you should be capable of writing some programs on your own. Have a go at the following short exercise. Don't forget to put in the 'RET' instruction, as otherwise the computer will trundle on through its memory after the program itself, looking for instructions to execute. It will almost certainly find something, and that something will cause the system to 'crash'. In other words, there will be no option but to switch off and start again from scratch.

EXERCISE 1.1

Using immediate mode addressing, load the accumulator with 65, and then display this on the screen. 65 is the ASCII code for capital A. A possible answer is given in the solutions chapter.

More About RST 08 and DEFB 158

If this call is used several times in succession, several things can be displayed on the screen in consecutive screen locations. Try this program:

PROGRAM 1.3

```
        ORG 256
        LD A,72          Load Accumulator with ASCII 'H'
        RST 8
        DEFB 158
        LD A,69          Load 'E'
        RST 8
        DEFB 158
        LD A,76          Load 'L'
        RST 8
        DEFB 158
        RST 8            Display second 'L'
        DEFB 158
        LD A,79          Load 'O'
        RST 8
        DEFB 158
        RET
```

That program should display the word 'HELLO' on the screen, after it has been assembled and run.

EXERCISE 1.2

Write your name on the screen. An answer for 'FRED' is given in the solutions chapter.

Summary

That's the end of this chapter. It wasn't that bad was it? By now you should know or be able to interpret the following:

O&4FØØ
LD A,n
LD r1,r2
RET
ORG
@
RST 8, DEFB 158
ADD A,n
Immediate mode addressing.
Register–register addressing.

How about this (r is any register)?

ADD A,r

Chapter 2

Jumping, Subroutines and Labels

Few real life programs proceed along a smooth uninterrupted path without jumping or branching at some stage. This chapter looks at these jump commands and their uses. After this, the chapter goes on to examine the flags that enable these jumps to be controlled. The chapter then looks at both unconditional and conditional calls to subroutines. Symbolic labels are also introduced. These are a very powerful and useful feature of the assembler and aid program development considerably.

Unconditional Jumps

These tell the program to jump willy–nilly – no conditions.

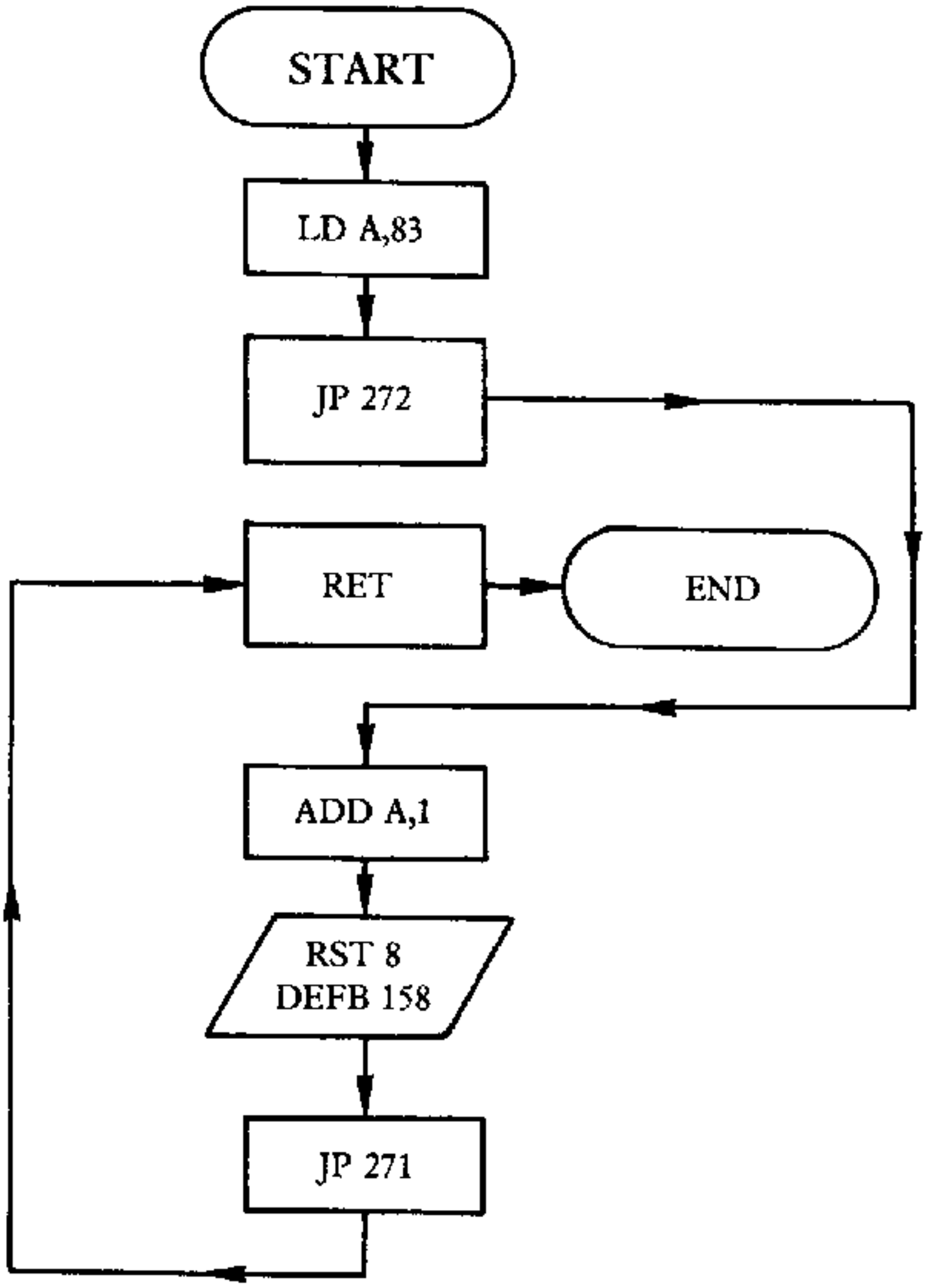
The Z80 instruction set contains five such jumps; for the moment we shall only be concerned with two of these. The other three will be introduced later.

The first to be considered is:–

JP nn	**Jum**P to the specified address **nn**

For example, 'JP 200' means 'jump to memory location 200'.

Put into a program, JP will look like this:



Using JP in a Program

FIGURE 2.1

Such a jump routine doesn't really achieve a lot but it could, for instance, be used to patch a piece of code into a program. In Figure 2.1 for instance, the commands 'ADD A,1', 'RST 08', 'DEFB 158' and 'JP 271' have effectively been inserted into the program.

We shall now type this program into the computer. Remember to type ENTER at the end of each line, and to choose 'O&4F00', selecting 'A' to start entering the program.

PROGRAM 2.1

```
ORG 256
LD A,83
JP 272
RET
ADD A,1
RST 8
DEFB 158
JP 271
@
```

Now Assemble the program with Z.

The screen display of the program should now look something like this:-

Memory Location	Line Number	Object Code	Source Code
C000	10		ORG 256
0100	20	3E53	LD A,83
0102	30	C31001	JP 272
0105	40	C9	RET
0106	50	C601	ADD A,1
0108	60	CF	RST 08
0109	70	9E	DEFB 158
010A	80	C30F01	JP 271

FIGURE 2.2

The first number in the memory location column doesn't seem to correspond to 256, where the program should begin. This is because the computer hasn't read the ORG 256 at that stage. The second number in the memory location column isn't 256 either, however. This is because memory locations are being represented in 'hexadecimal' numbers. Don't worry about this for now, this will be explained in Chapter 4 (or Appendix 5); for now, just remember that 0100 is 256, 0106 is 272 and 0105 is 271.

When jumps are used in this way it is necessary to tell the program where to jump to, i.e. an address, hence JP 272. Calculating these addresses is quite straightforward as long as it is done systematically, as will be explained below.

Looking at Figure 2.2, the numbers and letters in the third column, the 'object code', are what the assembly language looks like after it has been assembled – i.e. machine code. Each pair of alphanumeric characters (i.e. numbers and/or letters) in this column represents one item from the original assembly language. Thus, '3E' represents 'LD A' and '53' represents '83' (don't worry about why! It's hexadecimal and will be explained later). For numbers in assembly language bigger than 255, four machine–code alphanumerics are required. Thus 'C3' represents 'JP' and '1001' represents '272'. How does this help us calculate addresses to jump to? Well, each pair of digits fills up one memory location – one byte of memory. Thus, as it is known that the first instruction in the program is in memory location 256 (because of the 'ORG 256'), we can simply count pairs of machine–code alphanumerics.

For instance, to jump to the 'ADD A,1' instruction, we would need to jump to the memory location containing 'C6'. So, starting at 256 (0100), which contains '3E', we have 53, C3, 10, 01, C9, and then C6. That's 6 memory locations along from 256, or 272 (0106). Hence the program should jump to 272. Admittedly, before you've entered the 272 in the JP instructions, the '1001' won't be there for you to count. However, as you know that the address in machine code is going to be four characters (two bytes or memory locations), long, this is not a problem provided you write your programs out on paper before typing them in – which is good practice anyway.

The appendices contain tables detailing how many bytes each instruction requires to help in calculating jump addresses.

Here is a breakdown of the program:

ORG 256	Uses no memory, it is there for the assembler's use; it is called an 'assembler directive'.
LD A,83	Takes up 2 bytes the first is the object code for 'LD A', and the second is the data.
JP 272	Takes up 3 bytes – one for 'JP', i.e. C3, the other two are the memory location to jump to.
RET	Only requires one byte for its object code, C9.
ADD A,1	Requires 2 bytes. The first, C6, is the object code for 'ADD A', the second is the data.
RST 8 DEFB 158	These two instructions require 2 bytes: CF is the object code and 9E the ROM routine being called (158 in decimal).
JP 271	This is the same as the JP 272 except that the memory location to jump to is different.

By counting up the number of bytes we can calculate the length of the program: Program 2.1 is 13 bytes long.

Using the JP operator we can jump to any memory location from 0 to 65535. If the memory location to jump to is less than +129 or –126 bytes from the current memory location then Jump Relative can be used instead of the JumP instruction. The advantage of using this alternative instruction is *that, as it only requires two bytes to store as opposed to three for the JP* instruction, it is quicker.

JR e	**J**ump **R**elative to address **e**

For example, if you are at address 12000, and you want to jump ahead to address 12045, you would use

```
JR 12Ø45
```

You would not say JR 39: the computer will calculate the single-byte offset automatically. Remember, JR can only be used if you are jumping to an address within +129 or –126 of the memory location containing the 'JR' itself.

The Program Counter

Before going much further, the way in which a jump instruction accomplishes its task must be examined. Examining Figure 2.2 it can be seen that the program starts at memory location 256. In the body of the program we require it to jump to memory locations 272 and 271. How does the computer keep track of where it is? Inside the Z80 chip there is a '16 bit' register called the 'Program Counter' (PC). This register contains the address of the current instruction. When a machine code program is run, the program counter is set to the first memory location of the program. After execution of the first instruction, the program counter is updated, so that it points to the next instruction. Thus a jump instruction loads the PC with the address to jump to i.e: it 'points' it to the new instruction. The effect of this is that the program continues execution from instruction contained in the memory location pointed to by the jump address. Other instructions are available to alter the contents of the PC and these will be introduced when required.

Figure 2.3 illustrates the execution of the previous program.

Program	PC before execution	PC after execution
ORG 256	?	256
LD A,83	256	258
JP 272	258	272
ADD A,1	272	274
RST 8	274	275
DEFB 158	275	276
JP 271	276	271
RET	271	Back to Disc Operating System

FIGURE 2.3

So far the computer has jumped to another section of code unconditionally. Although useful, it would be more useful if the computer could be made to jump upon a certain condition being met. This can be accomplished with the conditional jump group of instructions.

Conditional Jumps

Any program that needs to test for conditions needs CONDITIONAL JUMPS. In BASIC, the analogy is the IF.... THEN command:

```
1Ø  IF X=Y THEN GOTO 5ØØ
```

This line causes the computer to compare the variables X and Y and if they are the same go to line 500.

The Z80 carries out this operation by using a special register, the flag register. The flag register is an eight–bit register like the accumulator and the B,C,D,E,H and L registers, but is used quite differently from these. Whereas the other registers are used to store and manipulate bytes, the flags register is treated as if it contained eight individual bits which are used as signals or flags. The Z80 normally only handles one flag at a time, either setting the bit value to 0 or 1. It can also test a flag to determine whether it is set (1) or reset (0).

For example, one of the flags is the Z flag or Zero flag. Whenever an arithmetic process is carried out that produces a result of zero, the zero flag is set to '1' – otherwise it will be reset to '0', indicating that the previous operation did not result in zero.

Several different instructions can set this flag, one of these being:

DEC d	**DEC**rement register **d**

This instruction decrements the contents of 'd', where 'd' is one of the following registers:

B, C, D, E, H, L, A

If, after decrementing the specified register, the result is equal to zero, then the zero flag is set, otherwise it is reset.

Note the instruction DEC is formally called an **OPERATOR**, and d, the **OPERAND**; thus the OPERATOR operates upon the OPERAND. Some operators, such as LD A,10 require two operands, in this case, A and 10.

Now on with the programming. Program 2.2 will print 10 A's on the screen.

PROGRAM 2.2

```
        ORG  256
        LD C,1Ø
        LD A,65
        RST 8
        DEFB 158
        DEC C              Decrement the C register
        JR NZ,26Ø
        RET                If zero then stop
```

Type this program in and assemble it using Z. To run the program exit to DOS with Q and type in a W with a filename you've chosen. When you run it, you will see ten A's appear upon the screen.

Examining the program, the only line which hasn't been met yet is JR NZ. This operator tests the current state of the zero flag and 'Jumps Relative' to 260 if the last arithmetic instruction results in a non–zero answer. Thus the program puts the contents of the accumulator upon the screen, DECrements register C and tests to see if the zero flag has been set by the DECrement instruction. If it hasn't, i.e. it is Non Zero, then it Jumps Relative to 260 otherwise it goes on to the next line of the program where it RETurns.

JR NZ,e	**J**ump **R**elative on **N**on **Z**ero result to address **e**

EXERCISE 2.1

Rewrite Program 2.2 to use the B register in place of the C register.

EXERCISE 2.2

Why have we used JR NZ,e instead of the alternative JP NZ,e instruction?

Possible answers to both exercises are given in the solutions chapter.

Note, as might be expected an instruction to INCrement an operand exists as well:

INC d	**INC**rement operand **d**

So far the computer has been programmed to jump unconditionally and on the result of an operation being zero. In both cases it is necessary to know which memory location to jump to. While this situation is bearable within short programs, when programs start increasing in size it becomes increasingly difficult to calculate these jump addresses, not to say time consuming. To overcome this problem, labels are used.

Labels

Labels allow a program to be directed to a named instruction without the programmer having to calculate jump addresses. A fancier term for label is SYMBOLIC LABEL as the label itself symbolises a location in memory.

For instance, the instruction

```
JP LOOP:
```

Will cause the assembler to replace 'LOOP:' with an address, which was previously assigned to 'LOOP:', whenever it occurs.

To tell the assembler that a label is a label it is necessary to follow the label with a colon. Also labels must be less than or equal to six characters in length. For example, if the program should jump to the memory location containing the instruction DEC C, the following program lines would be used.

```
        •
        •
        •
        •
JP LOOP:
        •
        •
        •
        •
LOOP: DEC C
        •
        •
        •
```

Further conventions must be observed when using labels. The colon for instance must follow the last character of the label (no space between them). Also, a space must follow the colon. This is simply to allow the assembler to work out where the label ends. Don't worry if you forget any of this, the assembler will pick up any errors and inform you.

Summary of Labels

1. A label must be six or fewer characters long.
2. A colon must immediately follow the label.
3. One space must follow the colon.
4. The label must not contain a space.

For example:

```
LOOP:

TEST:

NXTCHR:
```

are all valid labels. The following are not:

```
LOOP :

BACKONE:

NEXT L:
```

To illustrate the use of labels we will rewrite Program 2.1 using labels.

PROGRAM 2.3

```
        ORG 256
        LD A,83
        JP NXT:
END:    RET
NXT:    ADD A,1
        RST 8
        DEFB 158
        JP END:
```

When you list the above program, you will see that it is displayed with the labels out to the left of the main body of the program, like this, making it a lot easier to read:

```
        ORG 256
        LD A,83
        JP NXT:
END:    RET
NXT:    ADD A,1
        RST 8
        DEFB 158
        JP END:
```

To ease typing in programs they will be displayed in this format throughout the rest of the book. However, you should, of course, type them in in the normal way.

Using the instructions introduced so far it is possible to jump unconditionally or conditionally to anywhere in memory, or using a jump relative command to any location within +129 or –126 of the current position. Labels can also be used instead of calculating jump addresses.

All of these commands will now be combined to print the numbers one to nine on the screen. Remember in Chapter 1 we saw that the ASCII for T is 84? Well, the numbers one to nine also have a code associated with them. Look at the ASCII table in Appendix 7 and see if you agree that the code for '1' is 49 and that for '9' is 57.

Now to combine what has been learnt so far to solve the problem, i.e: how to put the contents of the accumulator on the screen, load it, and increment it. Also, how to decrement a register and test for zero. If you have any ideas about how to get the computer to display to numbers 1 to 9 on the screen, close the book now and try: if not, don't worry all will be explained.

The following flow chart represents the solution.

Problem: Display the numbers 1 to 9 on the screen.

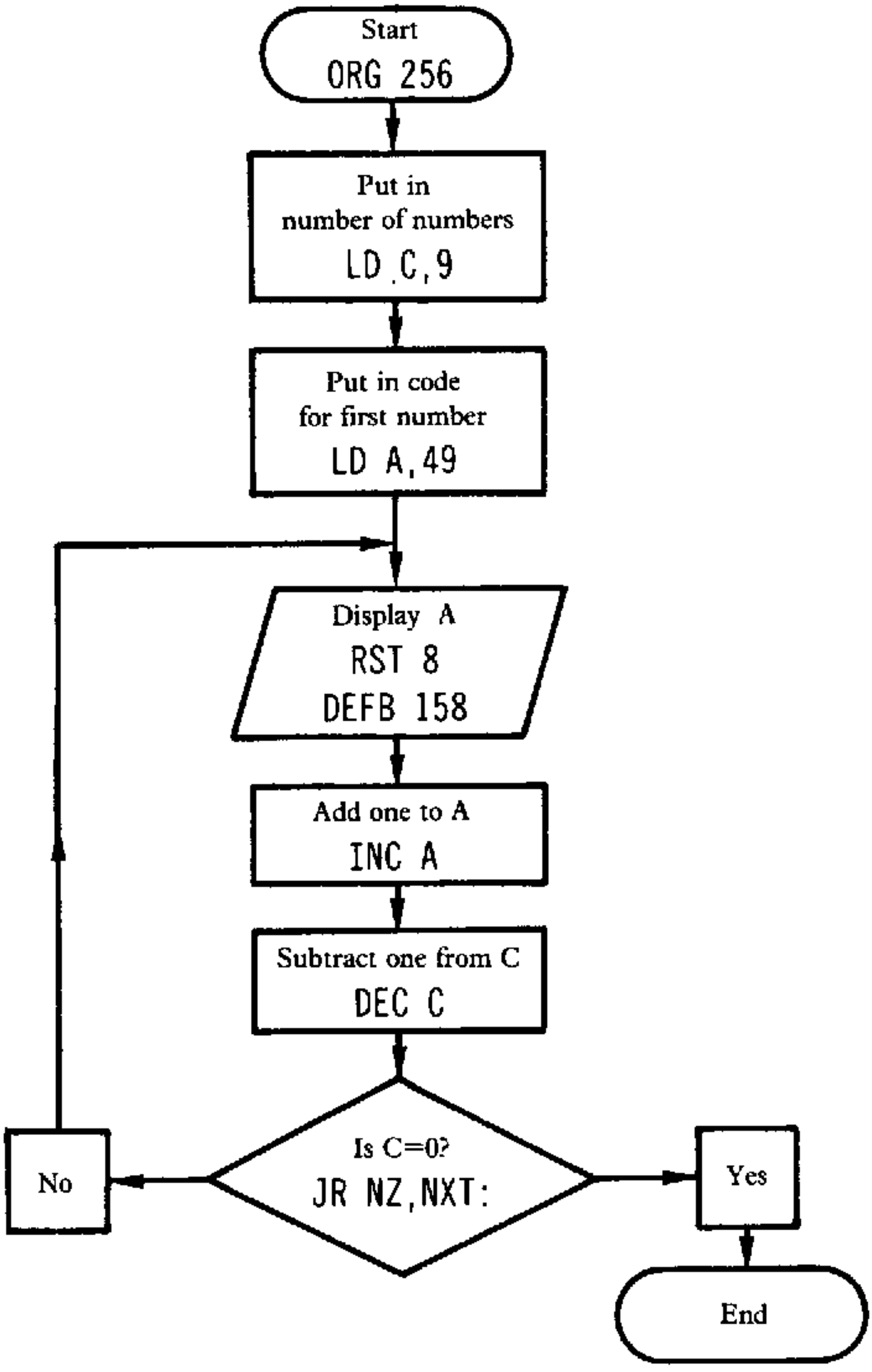


FIGURE 2.4

Translated into a program it looks like this:

PROGRAM 2.4

```
        ORG 256          Set start of program.
        LD C,9           Load C with number of characters
                         to print, i.e. nine.
        LD A,49          Load A with the code for '1'.
NXT:    RST 8            Print contents of Accumulator on
        DEFB 158         the screen.
        INC A            Add one to contents of A, thus
                         loading A with code for '2'.
        DEC C            Decrement the count register.
        JR NZ,NXT:       If DEC C results in anything but
                         zero jump relative to the location
                         addressed by the label NXT:.
        RET              If C=0 then return to DOS.
```

Before running it, let's step through it: each step is one loop of the program.

Step Number	Accumulator	Register C	Z flag
1	49	9	Ø
2	5Ø	8	Ø
3	51	7	Ø
4	52	6	Ø
5	53	5	Ø
6	54	4	Ø
7	55	3	Ø
8	56	2	Ø
9	57	1	Ø
1Ø	58	Ø	1

When the Z flag is set (1) the program terminates and returns to the DOS. (Step 10)

Now assemble and run the program.

EXERCISE 2.3

Write a program which will print the alphabet on the screen. (Hint: the code for A is 65.)

A possible answer is given in the solutions chapter.

So far only the zero flag has been used, one of the seven flags available. Here are the rest.

The Flags

The Carry Flag (C)

This flag is set whenever an addition or subtraction results in a carry (or borrow). It is also used in certain shift and rotate instructions.

Subtract Flag (N)

This flag is mainly used by the Z80 rather than by programs for certain arithmetic operations.

Parity/Overflow Flag (P/V)

This flag has two distinct functions, the first to indicate the parity of a result. The 'parity' of the result is obtained by adding up all the '1's in its binary representation. If the result is even then the parity but is set, otherwise reset, i.e. 0. The flag is also set when during certain arithmetic operations 'overflow' occurs.

The Half Carry Flag (H)

The half carry flag is used by the Z80 for 'Binary Coded Decimal' instructions. Don't worry about this for the moment.

Zero Flag (Z)

The zero flag is set when an operation results in zero, and as such, is used a lot for compares.

The Sign Flag (S)

This flag indicates the sign of the arithmetic result or of a byte being transferred. It basically tests bit 7 of a byte, and is set if it equals 1 and reset otherwise.

Subroutines

As the JP instruction was compared with the BASIC statement 'IF X=Y THEN GOTO 500', a Z80 instruction similar to the BASIC statement GOSUB 500 exists.

CALL nn	**CALL** the routine commencing at the memory address **nn** (nn can be represented by a label).

As in BASIC, a return instruction is required at the end of the subroutine.

RET	**RET**urn from the subroutine last called.

Let's see how these instructions can be used in a program.

Problem: Print the ASCII characters 200 to 250 on the screen using a subroutine print the characters.

PROGRAM 2.5

```
        ORG 256
        LD A,2ØØ          A=1st ASCII code.
        LD B,5Ø           B=count.
NXT:    CALL PRINT:
        DEC B
        INC A             Next ASCII code.
        JP NZ,NXT:        IF B=0 GOTO NXT:.
PRINT:  RST 8
        DEFB 158
        RET
```

Assemble then run it. Don't forget the colons (:) in your labels, otherwise the assembler will not assemble your programs properly.

To aid understanding, the print subroutine is assigned the label 'PRINT'.

As well as unconditional calls, conditional calls exist, much the same as conditional and unconditional jumps.

CALL cc,nn	**CALL** the subroutine starting at memory location **nn**, if condition **cc** is met.

The conditions on which the routine is called are exactly the same as those used for conditional jumps. When a CALL instruction is executed the current contents of the program counter are saved. The program counter is then loaded with the call address nn. When the Z80 has completed the subroutine, i.e. reached a RET instruction, the program counter is loaded with the saved value of the program counter.

Problem: Print 100 A's on the screen, but for every 10 printed, print a space.

Representing the solution as a flow chart:

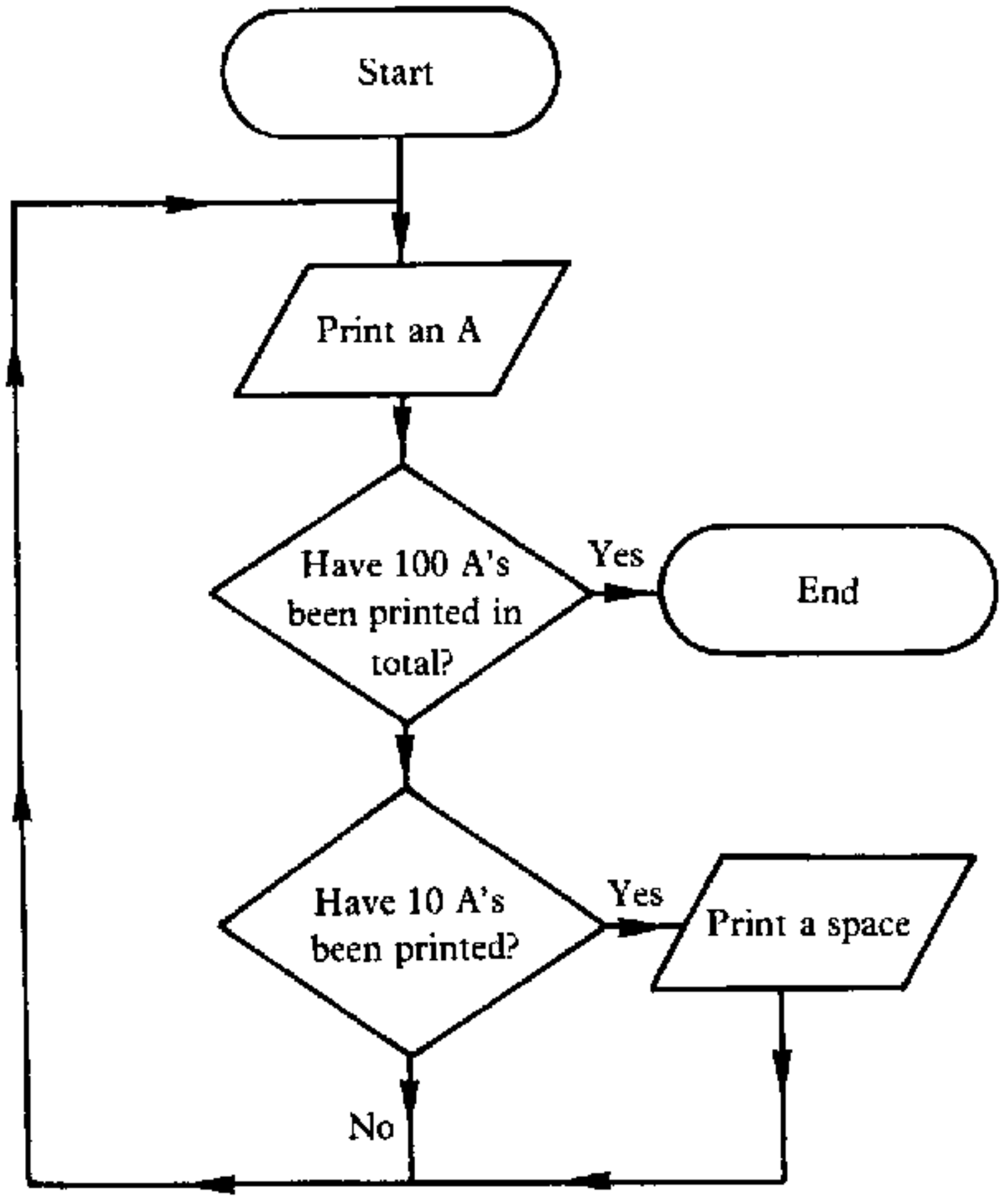


FIGURE 2.5

Converting this into a program:

PROGRAM 2.6

```
        ORG 256
        LD C,1ØØ        C— 1st count.
        LD B,1Ø          B  2nd count.
        LD A,65          A is "A".
NXT:    RST 8
        DEFB 158         Print "A".
        DEC B
        CALL Z,PSPC:     If B  0 print a space.
        DEC C
        JR NZ,NXT:       Have 100 A's been printed?
                         If not jump to label NXT:
        RET              Return to operating system.
PSPC:   LD A,32          32  ASCII code for a space.
        RST 8
        DEFB 158         Print the space.
        LD A,65          Remember to re–load the acc. with ASCII "A".
        LD B,1Ø          Reset the counter for spaces.
        RET              Return to main program.
```

Try running this program now; lo and behold a hundred A's appear upon the screen separated every 10 by a space. This form of conditional call is very useful as it considerably eases writing structured programs.

Summary

The program counter (PC) is a 16–bit register, which contains the memory address of the current instruction from within a program.

The flags register contains 8 bits, reflecting the status of the arithmetic section of the Z80.

Labels can be used in jump instructions. A label must be less than seven characters long, terminated by a colon which must be followed immediately by one space.

Two main forms of jumps exist, conditional and unconditional. These two forms can be divided further into absolute jumps, i.e. JP NXT: or relative jumps, i.e. JR NXT:.

Two forms of CALL exist, conditional and unconditional. Every call instruction must have associated with it a RET instruction.

The following list of instructions should be understood in context, even if you don't quite understand all the flag conditions.

```
JP nn
JP cc,nn
JR e
JR Z,e
JR NZ,e
```

Where:

nn is an absolute address or a label.

cc is one of the operands signifying which flag to test for.

e is an absolute address, except that it is limited to –126 to +129 bytes from the current address. A label can also be used.

cc	condition
NZ	Non–Zero
Z	Zero
NC	No–Carry
C	Carry
PO	Parity Odd
PE	Parity Even
P	Positive
M	Negative

Also:

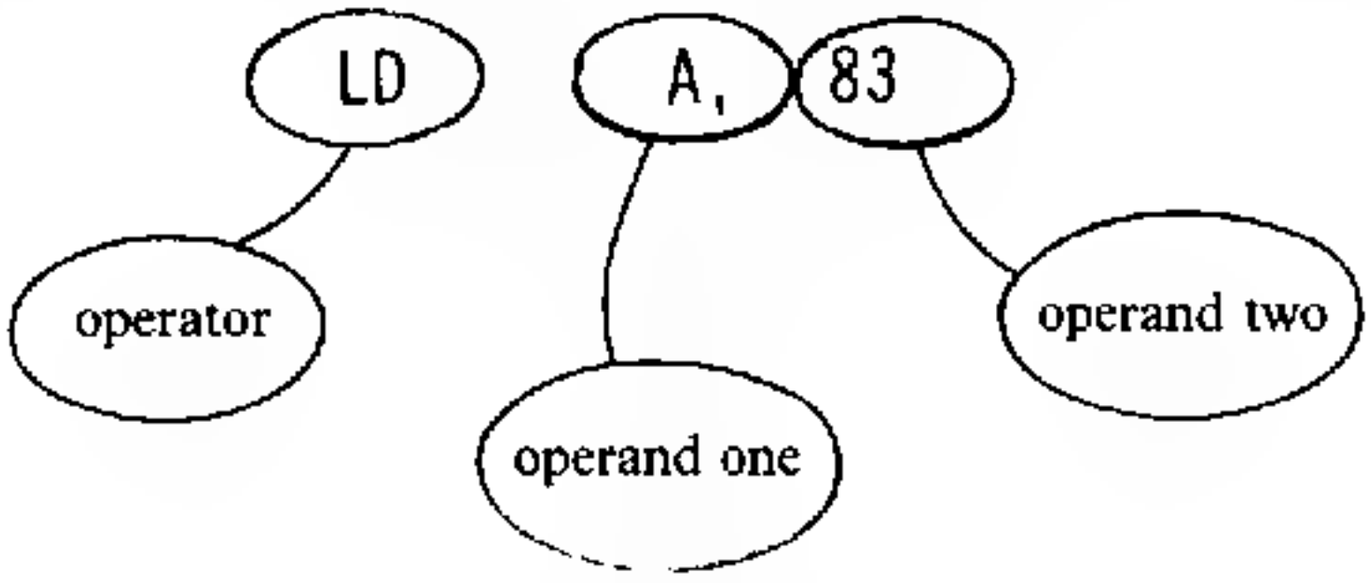
```
INC   r

DEC   r
```

Where r is an 8–bit register, any one of the following

A, B, C, D, E, H, L.

The definition of operand and operator is best shown with an example:



Chapter 3

Register Pairs And Addressing Modes

So far we have only been able to load a register with an 8–bit number. The Z80 allows certain registers to be combined into pairs, thus allowing '16–bit' numbers (i.e. numbers up to 65535) to be stored and manipulated.

The following registers may be paired together.

B and C
D and E
H and L

When loading numbers into register pairs, a part of the number goes in one register and the remainder goes into the other as will be seen later.

We have already met the instructions that enable us to load 8–bit data into individual registers. The instruction which allows 16–bit loading is:

LD dd,nn — **LoaD** register pair **dd** with the 16–bit data **nn**.

Where dd is any of the following registers: BC, DE, HL, or SP. 'nn' is a 16–bit number.

Note: The SP register is a special–purpose register, called the stack pointer. We will return to it at a later stage.

When loading a register pair in this way, we are loading it immediately with the data value given; hence the formal term for this type of addressing: **IMMEDIATE MODE.**

All the programs so far have stored data in registers. This situation is fine if we don't have very much data to process. In real–life situations though, we will require more data storage than the seven registers so far described.

A need arises for intructions to load memory with data and to read the contents of a memory location. The Z80 allows the contents of registers to be stored in memory and the contents of memory locations to be copied into registers.

This form of addressing is termed **DIRECT MODE,** because it uses the contents of memory locations and registers directly – i.e. without doing anything with them first.

LD (nn),dd	**LoaD** memory location **nn** with the contents of register pair **dd.**

An example of this instruction is:

```
LD (2ØØ),BC
```

This will copy the contents of register pair BC into memory location 200. Note that the memory location has to be given in brackets.

The opposite instruction, to copy the contents of a memory location into a register pair is:

LD dd,(nn)	**LoaD** register pair **dd** with the contents of memory location **nn.**

An example of this instruction is:

```
LD BC,(2ØØ)
```

What will this do? It will load the contents of memory location 200 into BC.

OK, let's now use these instructions.

Up until now characters have been printed on the screen with a RST 08 instruction followed by MCAL (Machine–code CALl) 158 (DEFB 158). This is not the only built–in routine available. The internal memory (ROM) also contains some graphics routines, which among other things allow us to draw lines on the screen. We will not be concerned with how this is accomplished, but with how to use it. When in BASIC, to draw a line from the origin to the point 200,200, the following command would be typed in (assuming that the graphics cursor is at 0,0)

```
DRAW Ø,Ø TO 2ØØ,2ØØ
```

If you try this, ensure that you reset the graphics cursor to 0,0 with the following command:

```
ORIGIN Ø,Ø
```

By using the built–in machine calls we can plot a line on the screen. The method of doing this may seem somewhat complex, but will become more meaningful as your familiarity with assembly language programming and the Einstein's MCAL's grows. Let us now summarise the information required to use the MCALS in order to plot a line on the screen.

We must first come to understand the meaning of the computer buzzword 'scratchpad'. As its name suggests, the scratchpad is an area of memory set aside for the storage of temporary information. When we plot lines on the Einstein's screen, we use certain parts of the scratchpad to hold the line type, start and end positions of the line.

Scratchpad Locations Used

64424	Length to end of first line
64425	Length to end of first space
64426	Length to end of second line
64427	Length to end of second space

Normally the values in these four locations will be in ascending order, but for a continuous line, location 64424 should be set to 255 and the other locations set to zero. BE WARNED... if you do not set location 64424 to 255 and the next three locations to zero, then your program could hang up leaving you with no alternative but to switch off.

If this seems somewhat overcomplicated, do not feel alone, this author felt that way too at first. Just think of it as though location 64424 were a switch being turned on by setting it to 255. The other three 'switches' are turned off by setting their contents to 0.

The other information required to draw the line is given below. Note that two new registers are used: IX and IY. These are called the 'index registers' and will be explained in more detail later in this chapter. The reason they are used here is simply that the line–drawing routine expects them to be used.

a) The starting X coordinate (Held in the IX register)

b) The starting Y coordinate (Held in the IY register)

c) The finishing X coordinate (Held in scratchpad location 64406)

d) The finishing Y coordinate (Held in scratchpad location 64408)

The actual MCAL or built–in subroutine required to draw the line is MCAL 200.

Thus to summarise:–

We have decided to draw a solid (i.e. continuous) line from 0,0 to 200,200.

Thus, before we start we must set up the appropriate scratchpad locations with the relevant data.

LINE TYPE

```
LD (64424),255
LD (64425),Ø
LD (64426),Ø
LD (64427),Ø
```

Switch solid line mode on.

Switch all other 'switches' off.

START POINT

```
LD IX,Ø
LD IY,Ø
```

END POINT

```
LD (644Ø6),2ØØ
LD (644Ø8),2ØØ
```

End X coordinate

End Y coordinate.

DO THE MCAL

```
RST 8
DEFB 2ØØ
```

MCAL to draw line.

So let's now convert this into a program.

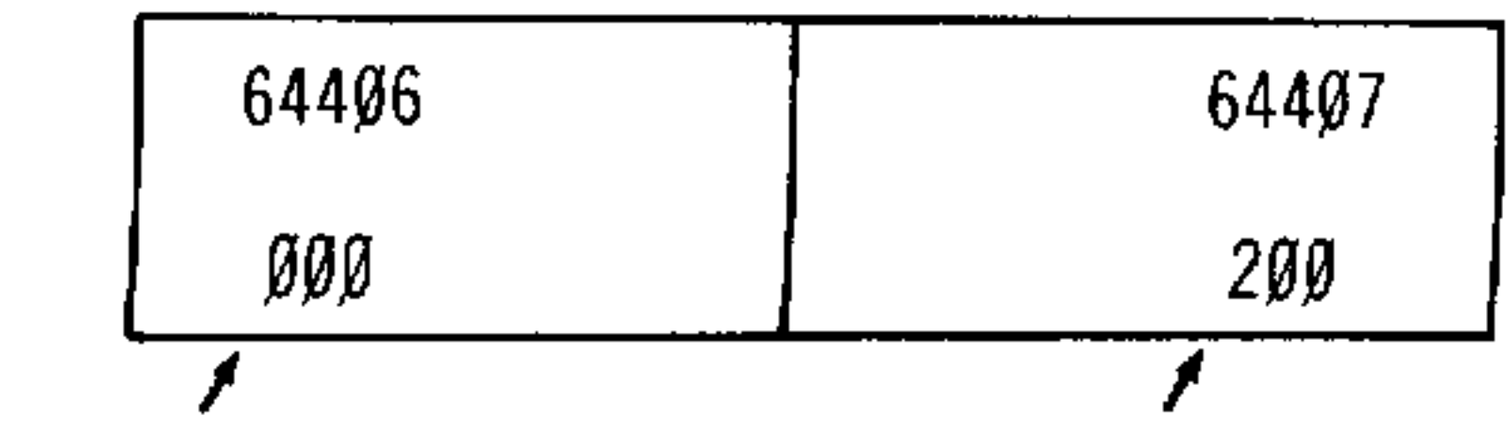
PROGRAM 3.1

```
        ORG 256
        LD A,255
        LD (64424),A        set line type.
        LD IX,0
        LD IY,0
        LD A,200
        LD (64406),A        X for end.
        LD (64408),A        Y for end.
        LD A,0
        LD (64425),A
        LD (64426),A
        LD (64427),A
        RST 8
        DEFB 200
        RET
```

After successful assembly, save the program to disc and then execute it directly from DOS (i.e. by typing in its name). You should get a diagonal line running from the bottom left hand corner of your screen. If the Einstein 'hangs', check that you have initialised ALL the scratchpad locations as discussed above. Incidentally, rather than having to type in the whole program again, you can save the original, unassembled version to disc as well as the assembled object code. See Appendix 6 for details of the 'S' command.

Note that the END locations for the X and Y coordinates are stored at 64406 for the X and 64408 for the Y. Why not store the Y coordinate at 64407? The reason is that one memory location can only store an 8–bit number and we are trying to store a 16–bit number (i.e. a number which can be greater than 255).

See Figure 3.1



The accumulator is holding the value 200 (in truth we must load 00200 into 64406 and 64407).

Memory	Contents	
.	.	
.	.	
.	.	
64Ø6	2Ø	
64Ø7	Ø	← Set by default on Einstein
.	.	
.	.	

Figure 3.1

The 'Least Significant Byte' (i.e. the rightmost part of the number 00200) is stored FIRST, followed immediately by the 'Most Significant Byte', which in our particular case is 00, and is already set as such by default in the Einstein on Boot–Up.

The Least Significant Byte is often abbreviated to LSB and the Most Significant Byte to MSB.

EXERCISE 3.1

Write down the order in which the registers would be stored if the following sub–program was executed. Note: when paired, the C, E and L registers are classed as the Least Significant Bytes, hence the Most Significant Bytes are registers B, D and H.

```
LD (2ØØ),DE
LD (2Ø2),HL
```

See the solutions chapter for the answer.

EXERCISE 3.2

Write a program to plot a line from 0,0, to 100,200

Answer in the solutions chapter.

EXERCISE 3.3

Write a program which will join the following points together with lines:

X	Y
2ØØ	3ØØ
4ØØ	2ØØ
Ø	Ø

A possible solution is given in the solutions chapter.

The Z80 chip also supports various other addressing modes. So far, the immediate mode for both 8 and 16–bit data, as well as direct mode for 16–bit data have been used. Another mode is Indirect.

Indirect Mode Addressing

In this mode of addressing, the contents of a register are used to point to a memory location. This mode is very useful, as we can load a 16–bit register with the starting memory address of our data, read the data, then increment (i.e. add 1 to) the register pair and access the next item of data.

The instruction family is as follows:

LD r,(HL)	**LoaDs** register **r** with the contents of memory location pointed to by **HL**. (This in effect means that we load HL with the memory address).

LD (HL),r	**LoaDs** the memory location pointed to by **HL** with the contents of register **r**.

LD A,(BC)	**LoaDs** the accumulator with the contents of memory location pointed to by **BC**.

LD A,(DE)	**LoaDs** the accumulator with the contents of memory location pointed to by **DE**.

LD (BC),A	**LoaDs** the memory location pointed to by **BC** with the contents of **A**ccumulator.

LD (DE),A	**LoaDs** the memory location pointed to by **DE** with the contents of the **A**ccumulator.

The above table may appear formidable at first, hopefully not, anyway, all will be explained as the instructions are used in examples.

What we will do now, to illustrate the use of some of the above commands, is to store the ASCII representation of three A's in memory, then print them on the screen using the HL register as a 'pointer'.

To simplify loading the three A's into memory we will use BC as a moving pointer. Thus, if the accumulator contains the code for 'A' (65), then using LD (BC),A and INC BC we can load three A's into consecutive memory locations.

Here is a program to do this:

PROGRAM 3.2 (don't assemble this yet)

```
ORG 256
LD BC,35ØØØ      BC - start of data in memory.
LD A,65          A=65, the ASCII code for 'A'.
LD (BC),A        Store first A.
INC BC           Increment pointer.
LD (BC),A        Store second A.
INC BC           Increment pointer.
LD (BC),A        Store third A.
```

Now to write the program to read the data and display it on the screen.

```
LD HL,35000          HL-start address.
LD A,(HL)            A-contents of memory location addressed by HL.
CALL PRINT·          Print contents of A on screen.
INC HL               Increment pointer.
LD A,(HL)            Re-load A.
CALL PRINT:          Print second character.
INC HL               Increment pointer.
LD A,(HL)            Re-load A.
CALL PRINT:          Print third character.
RET                  Return.
PRINT: RST 8
DEFB 158
RET
```

Assemble and run this program now. While this program accomplishes its aim, it is not very efficient. Let's now see how we can improve upon it.

Considering the first section:-

We duplicate the following commands three times, to set up the data.

```
LD (BC),A
INC BC
```

Why not set up a loop? If we load an unused register, say E, with 3, then every time we load a memory location, we decrement E and test it to see if this results in a zero. If so, we end the loop, otherwise we jump back to the beginning of the loop. This method accomplishes the same task with greater elegance and efficiency.

Here is the first section of the modified program:

PROGRAM 3.3 (Don't assemble this yet)

```
        ORG 256
        LD BC,35ØØØ
        LD A,65
        LD E,3          Load count register.
NXT:    LD (BC),A
        INC BC
        DEC E           Decrement count.
        JR NZ,NXT:      If not zero jump back to NXT.
```

Note the use of the label NXT to simplify the calculation of the jump address.

No great time is saved in typing this program in: imagine though using the previous method to store 200 'A's!

The same techniques can be applied to the second section.

EXERCISE 3.4

Add a second section to the program, using a loop to print out the three A's, then assemble and run it.

EXERCISE 3.5

Write a program to store the alphabet in memory, then read it out of memory onto the screen.

See the solutions chapter for possible answers.

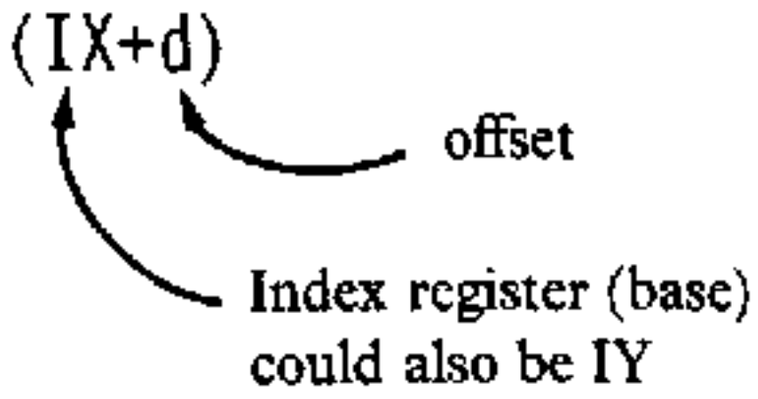
Let's briefly recap upon the addressing modes used so far.

Register –Register	**Immediate**
LD B,A	LD A,83 LD BC,300
Direct	**Indirect**
LD A,(2000)	LD B,(HL)

You will find a full list of the Z80 mnemonics in the appendices. Most of the 8 and 16 bit load groups should be comprehensible by now.

Indexed Addressing

In Program 3.3 we used a register pair to access a sequentially stored block of data. While in certain applications this method is very efficient, the Z80 offers an alternative. In addition to the registers already introduced, the Z80 contains two 16 bit 'index registers', IX and IY, which you have already used without a proper explanation of their purpose. If you look up the word 'memory' in the index of this book, you will probably start by finding the beginning of the M's, then searching for the word 'memory'. In other words you have treated the start of the M's as a base and then continued your search from there. This is directly analogous to indexed addressing. The IX or IY register is loaded with a base address. Then, to access a piece of information, an offset is added to this base. The general format for the indexed part of an instruction is as shown below:

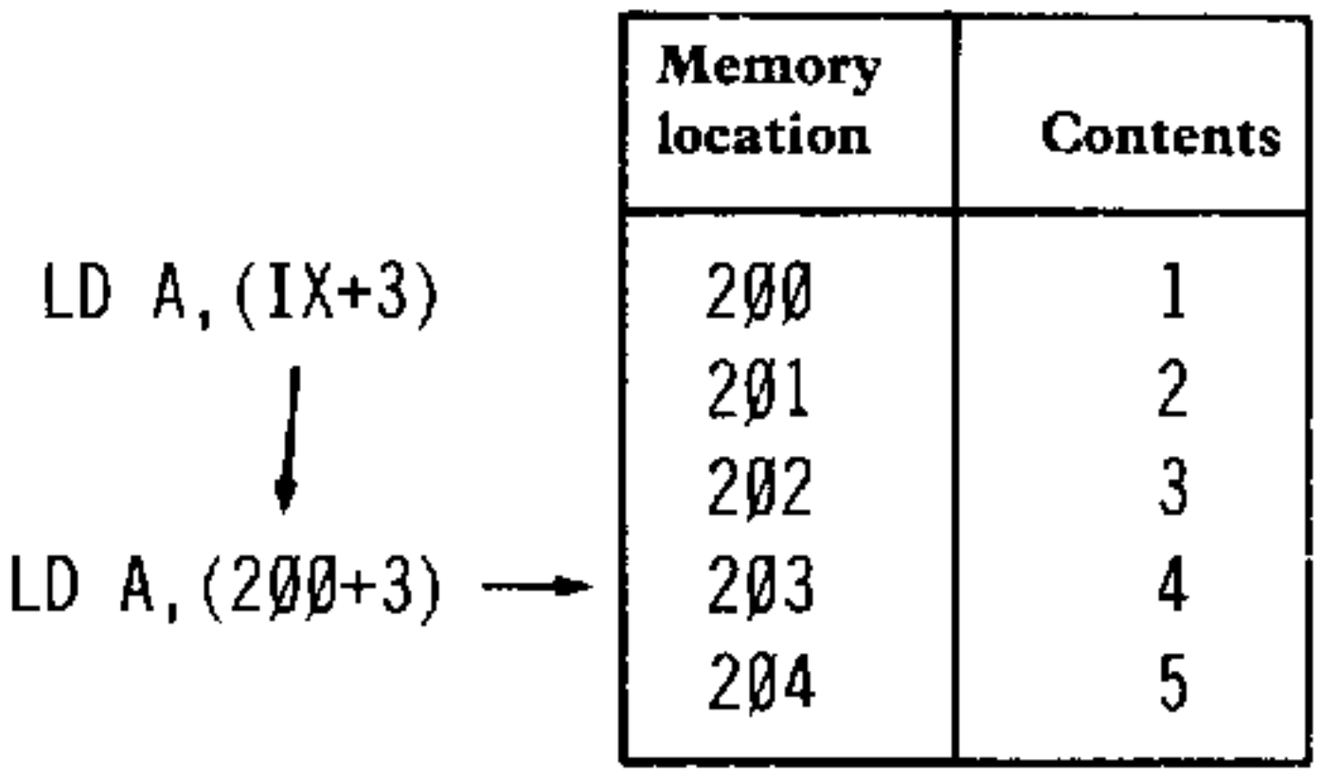


'd' is a number in the range –127 to +128. Here is a sample instruction using the index register IX:

```
LD A,(IX+3)
```

What will this do?

Suppose IX contains 200. What value will be loaded into A?



Memory location	Contents
200	1
201	2
202	3
203	4
204	5

FIGURE 3.2

The accumulator will therefore contain 4 after this instruction. What offset would you have had to use to load A with 3? Answer: (IX+2)

Instead of using the IX register as a base we could have used IY with the instruction:

```
LD A,(IY+3)
```

Now let's review the load instructions which use either IX or IY.

Indexed Addressing Instructions

There is a shorthand method of describing the operation of an instruction, referred to as its symbolic operation. Here is an example:

Instruction	Symbolic Operation
LD r,(IX+d)	r ← (IX+d)

This loads register r with the contents of the memory location pointed to by the addition of IX and d. The appendices contain a complete table of Z80 instructions including symbolic operations.

Here are the indexed loading instructions:

Instruction	Symbolic Operation
LD r,(IX+d)	r ← (IX+d)
LD r,(IY+d)	r ← (IY+d)
LD (IX+d),r	(IX+d) ← r
LD (IY+d),r	(IY+d) ← r
LD (IX+d),n	(IX+d) ← n
LD (IY+d),n	(IX+d) ← n

Now let's see how we can load the IX or IY registers themselves. Look up the 16 bit load group table, and see if you can identify *what instructions to use.*

Here they are; the format should appear familiar:

Instruction	Symbolic Operation
LD IX,*nn*	IX ← *nn*
LD IY,nn	IY ← nn
LD IX,(nn)	IX ← (nn)
LD IY,(nn)	IY ← (nn)
LD (nn),IX	(nn) ← IX
LD (nn),IY	(nn) ← IY

One point to note: it is not possible to compute the value of 'd', it has to be given absolutely – i.e. as an immediate value.

To illustrate the use of index registers, Program 3.2 has been rewritten.

Note: From now on the ORG instruction will be omitted from all programs: it will be assumed that you will automatically insert it as the first line.

Here is the program:

PROGRAM 3.4

```
LD IX,35ØØØ          IX  base
LD A,65
LD (IX+Ø),A          First 'A' stored.
LD (IX+1),A          Second 'A' stored.
LD (IX+2),A          Third 'A' stored.
```

The second section, which prints out the contents of memory on the screen, is as follows:

```
LD A,(IX+Ø)
CALL PRINT:          Print first character.
LD A,(IX+1)
CALL PRINT:          Print second character.
LD A,(IX+2)
CALL PRINT:          Print third character.
RET
PRINT: RST 8
DEFB 158
RET
```

Because the Z80 has to add the offset to the base for indexed instructions, it requires more time to execute than an indirect or immediate instruction. Also, because 'd' has to be given as an absolute value, indexed addressing is not much good in a loop, as 'd' cannot be incremented. However, the main advantage of IX and IY lies in their ability to access data, which is, stored in memory relative to some known memory location – i.e. for accessing tables of data, where various items of information are stored using known offsets.

EXERCISE 3.6

Write a program using IX to store five letters, e.g. a name. Then print the third and fourth letters on the screen.

The following table may help.

Memory location	Letter	Code for letter
35ØØØ	S	83
35ØØ1	U	85
35ØØ2	S	83
35ØØ3	A	65
35ØØ4	N	78

A possible solution is given in the solutions chapter.

Well done! The end of another chapter; all that is left now is a summary of what you have learnt during this chapter. If you feel up to it, try writing your own programs. How about printing a character, erasing it by printing a space over it, then printing it in another location. This could be the basis of a game!

Summary

The following terms should now be understood:

1. Register Pairs

2. Addressing Modes

    a) Register–Register
    b) Immediate

c) Direct
d) Indirect
e) Indexed

3. M.S.B. and L.S.B.

4. Symbolic Operation

5. Although not specifically mentioned, the following commands will probably make sense (ss is a register pair).

```
INC ss
DEC ss
```

6. You should now be able to understand these three instructions which were mentioned briefly in Chapter 2:

```
DEC (HL)
DEC (IX+d)
DEC (IY+d)
```

Where ss is any one of the following (ignore SP for now)

BC,DE,HL,SP

d is a number in the range –126 to +129.

Chapter 4

Arithmetic Operations

Most real life situations require the manipulation of numbers. So far, we can only increment and decrement the contents of registers or memory locations. The Z80 provides us with some additional arithmetic instructions which although not extensive form the basis of more powerful operations.

This chapter explains these instructions. If you are not familiar with the binary or hexadecimal representation of numbers turn to Appendix 5 and work through it.

Welcome back!

The concepts of binary and hex (hexadecimal) should now be familiar. You will see why we use hex soon.

We can classify the arithmetic instructions into two groups: 8 and 16 bit. In this chapter we shall be concerned mainly with the 8–bit group, as the 16–bit group naturally follows on.

The most common arithmetic process is adding two numbers. To add two 8–bit numbers the following instruction is used:

ADD A,n	**ADD n** to the contents of the **A**ccumulator, replacing the contents of the accumulator with the result.

To illustrate this command we will add 65 and 20 giving 85. When the result is printed on the screen we will see a U. (The character corresponding to the ASCII value of 85).

Try the following program.

PROGRAM 4.1

```
LD A,65          Load A with 65
ADD A,2Ø         ADD 20 to A
RST 8    }
DEFB 158 }       Print the result on the screen
RET
```

Lo and behold a U appears on the screen.

The assembler also allows numbers to be represented in hexadecimal. To allow the assembler to distinguish between them, hexadecimal numbers are preceded by an ampersand, '&'.

Rewriting Program 4.1 using the hexadecimal representation for 256, 65, 20 and 158:

PROGRAM 4.2

```
LD A,&41
ADD A,&14
RST 8
DEFB &9E
RET
```

Verify that this program is exactly the same as Program 4.1 by running it.

EXERCISE 4.1

Write a program which will add 200 and 48 together then print the result on the screen.

EXERCISE 4.2

Write a program that will add &41 and &10 together. Then print the result on the screen. What will be printed?

Possible answers are given in the solutions chapter.

So far all our answers have been less than 255, the maximum number an 8 bit register can hold. What do you think will happen if 150 and 171 are added together? Try the following program.

PROGRAM 4.3

```
LD A,15Ø        Load A with 150
ADD A,171       ADD 171 to A
RST 8     }
DEFB &9E  }     Print the result on the screen
RET
```

What has happened is that the accumulator has overflowed. It reached 255 then when 1 more was added it went back to zero and started again, reaching 65; hence the 'A' on the screen.

What cannot be seen from the program is the contents of the flags register. If observable, it would be seen that as the accumulator overflowed the carry bit was set (1). This fact can be used to add together two numbers whose sum is greater than 255. This process will be illustrated on paper first, then a program will be written to accomplish the same task in assembly language.

Problem: Add 1157 to itself. i.e: $1157 + 1157 = ?$

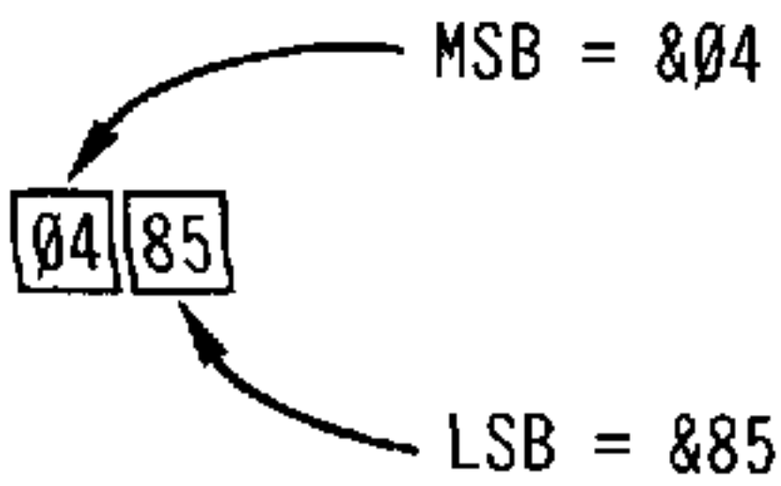
Step 1

Convert 1157 to Hex.

```
1157 ÷ 4Ø96 = Ø remainder 1157
1157 ÷ 256  = 4 remainder 133
133  ÷ 16   = 8 remainder 5
5    ÷ 1    = 5 remainder Ø
```

Thus 1157 = &0485.

&0485 expressed in binary is a 16–bit number. This therefore needs to be split in half to allow the use of 8–bit arithmetic instructions. This is easily accomplished in hex. (Which is why it is used!)



Note the terms MSB and LSB, meaning Least and Most Significant Byte, respectively.

Step 2

To add the two &0485's together we must first add the LSB's then add the MSB's, taking into account any carry generated by the addition of the LSB's.

```
   85
  +85
+carry ØA
```

```
    8  5
   +8  5
  ------
+carry 6 1Ø   Decimal
+carry Ø  A   Hex
```

Thus the result is &0A + a carry.

Step 3

Add the MSB's taking the carry bit into account.

```
  Ø4
+Ø4
---
  Ø8
```

Now add the carry bit:

```
  Ø1  ←  carry bit
+Ø8
---
  Ø9
---
```

Step 4

Recombine the new LSB and MSB.

```
1157 + 1157 = &Ø9ØA
```

Now satisfy yourself that &090A equals 2314_{10}.

The main point to note is that in all DOUBLE PRECISION work, i.e. using 16--bit numbers, the LSB is always operated upon first, so that the carry (if any) can be taken into account.

Before we can write the program to accomplish this addition we have to be aware of some new instructions:

AND A	Logical **AND** of the **A**ccumulator.

Only one effect of this operation needs to be observed at the moment, which is resetting the carry flag to zero. Why is this necessary? Answer: if the carry flag was unintentionally set, the result after the addition would be incorrect due to the inclusion of the incorrect carry in the addition.

An instruction which adds two registers plus the contents of the carry flag is now required.

ADC A,s	**AD**d the contents of register **s** plus the **C**arry flag to the contents of the **A**ccumulator. The result is stored in the accumulator.

Note that the ADC instruction uses a register as one of its operands. (The meaning of 's', 'r' and other abbreviations is given in Appendix 1.) An instruction similar to ADD A,n exists using a register in place of the immediate data n:

ADD A,s	**ADD** the contents of register **s** to the contents of the **A**ccumulator. The result is stored in the accumulator.

Now on to the program.

PROGRAM 4.4 (don't assemble this yet)

```
LD C,&85          Load C with the 1st LSB.
LD A, &85         Load A with the 2nd LSB.
AND A             Clear the carry flag.
ADD A,C           A = LSB + LSB.
LD (&7000),A      Save new LSB.
LD C,4            Load C with the 1st MSB.
LD A,4            Load A with the 2nd MSB.
ADC A,C           A = MSB + MSB + carry.
LD (&7001),A      Save new MSB.
```

Well this program will add the two numbers together. But how can the result be checked? What is now needed is a program to display the answer in a recognisable format. Why can't the contents of the two memory locations be printed on the screen? Nothing recognisable would happen, as the ASCII codes &09 and &0A do not represent characters but control codes. An offset has to be added to both answers to bring them into the ASCII alphabet range (65–122).

As the code for A is 65, this seems a reasonable offset to use. Thus adding the following lines to the previous program from where we left off:

PROGRAM 4.4(a)

```
        LD C,65
        LD A,(&7ØØ1)
        ADD A,C
        CALL PRINT:
        LD A,(&7ØØØ)
        ADD A,C
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

Now assemble and run the whole program. You will see a J and K on the screen (corresponding to 9 + 65 and &A + 65).

EXERCISE 4.3

Write a program to add together 250_{10} and 600_{10}, in hexadecimal, adding 65_{10} to the MSB and LSB. Then print the result on the screen.

Subtraction

SUB s	**SUB**tracts the contents of register **s** from the accumulator. The result is stored in the accumulator.

SBC A,s	**SUB**tracts the contents of register **s** plus the **C**arry flag from the **A**ccumulator. The result is stored in the accumulator.

The process of subtracting two 8–bit numbers is left for Exercise 4.4. Try it!

EXERCISE 4.4

Write a program that will subtract 9 from 233. Then print the result on the screen. Note that there is no need to add the offset of 65 to the result. Why?

EXERCISE 4.5

Write a program that will calculate the answer to the following sum:

$$(97 + 126) - 153 = ?$$

A possible answer is given in the solutions chapter.

As seen previously, when adding two numbers that give a result greater than 255, overflow occurs. This overflow causes the carry flag to be set.

The opposite situation occurs when trying to subtract a large number from a smaller one. For example, try the following:

```
  25
-17
 ---
  ?
 ---
```

It is not possible to directly subtract 7 from 5 so a unit is borrowed from the next column. The first step in the subtraction now becomes:

```
   5 + borrow = 15
 - 7            - 7
 ---            ---
   ?              8     Ans1=8
 ---            ---
```

As we have borrowed we have to 'pay back' i.e: subtract one from the next column.

Completing the subtraction:

```
   2 - borrow =  1
 - 1            - 1
 ---            ---
   ?              Ø     Ans2=0
 ---            ---
```

Combining the two answers (Ans1 and Ans2)

```
Ø + 8 = 8
```

Thus:–

```
27 - 17 = 8
```

By using the same process we can perform this subtraction using the Z80 subtract instructions.

Problem: Subtract 2000 from 2224.

Step 1

Convert the two numbers to Hex.

```
2000   4096 = 0     remainder 2000
2000   256  = 7     remainder 208
208    16   = D     remainder 0
0      1    = 0

2224   4096 = 0     remainder 2224
2224   256  = 8     remainder 176
176    16   = B     remainder 0
0      1    = 0

2000  = &07D0

2224  = &08B0
```

Step 2

Subtract the two LSB's

```
   B0 + borrow = 1B0
 - D0                  D0
 ----                 ----
   ??                 E0
 ----                 ----

                      0=0-0
                      E=1B-D
```

Step 3

Now the MSB's

```
   Ø8                  Ø8
- Ø7   - borrow =-Ø8
??                  ØØ
```

Combining the two results:

```
2224-2ØØØ = &ØØEØ
```

When carrying out this form of subtraction the carry flag acts as a borrow flag, being set when a borrow occurs. Let's now convert this subtraction into Z80 instructions.

PROGRAM 4.5

```
        LD C,&DØ          C– 1st LSB
        LD A,&BØ          A= 2nd LSB
        AND A             Clear carry flag
        SUB C             A= new LSB
        LD (&7ØØØ),A      Save LSB
        LD C,7            C= 1st MSB
        LD A,8            A= 2nd MSB
        SBC A,C           A–A–C–carry
        LD (&7ØØ1),A      Save MSB
        LD A,(&7ØØØ)      Recall LSB
        CALL PRINT:       Print result
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

Points to note: firstly, as the MSB of the answer is known to be zero there is no need to print it. Secondly, there is no need to add any offset to the answer as it is within the printable range of ASCII characters (65–255). Character 224 will be printed (diagonal stripes).

EXERCISE 4.6

Write a program to perform a 16–bit subtraction using the following instruction (ss is a register pair – i.e. BC or DE in this case).

```
SBC HL,ss
```

eg. 4248 – 4008 = ?

(Hint: it is a lot easier than Program 4.5.)

Answer in the solutions chapter.

EXERCISE 4.7

Store two numbers in memory, then add the contents of the accumulator to both. Replace the old values with the new calculated results.

The following tables may help.

Memory location	Contents
&7000	10
&7001	20

Accumulator = 65

After program has been run

Memory location	Contents
&7000	75
&7001	85

Use the instruction ADD A,(HL). Verify that the results are as expected by printing the contents of the memory location on the screen.

Answer in the solutions chapter.

That concludes the basics of addition and subtraction. Although not explicitly mentioned, the following instructions should be self–explanatory.

Instruction	Symbolic Operation
ADD HL,ss	HL ←— HL+ss
ADC HL,ss	HL ←— HL+ss+carry
SBC HL,ss	HL ←— HL−ss−carry
ADD IX,pp	IX ←— IX+pp
ADD IY,rr	IY ←— IY+rr

Where:

ss is any one of BC,DE,HL or SP
pp is any one of BC,DE,IX or SP
rr is any one of BC,DE,IY or SP

You should be able to understand what each instruction accomplishes from the symbolic operation. It is possible to add or subtract 32–bit numbers using the above instructions and the carry flag using the same method by which 16–bit arithmetic was carried out using 8–bit instructions.

The Carry Flag Instructions

Before any arithmetic operations, the carry flag has to be reset. This process has been accomplished using the AND A instruction. The carry flag can also be reset by using the two following instructions:

SCF	**S**et the **C**arry **F**lag

CCF	**C**omplement the **C**arry **F**lag

To reset the carry flag with these instructions, firstly the carry flag is set using the SCF instruction, then it is complemented by using the CCF instruction. When a binary number is complemented, a zero is replaced by a one; likewise a one is replaced by a zero. The carry flag when set will equal one, so, after complementing it, it will equal zero. The reason AND A was used earlier in this chapter is that it requires half the time required by SCF and CCF.

Summary

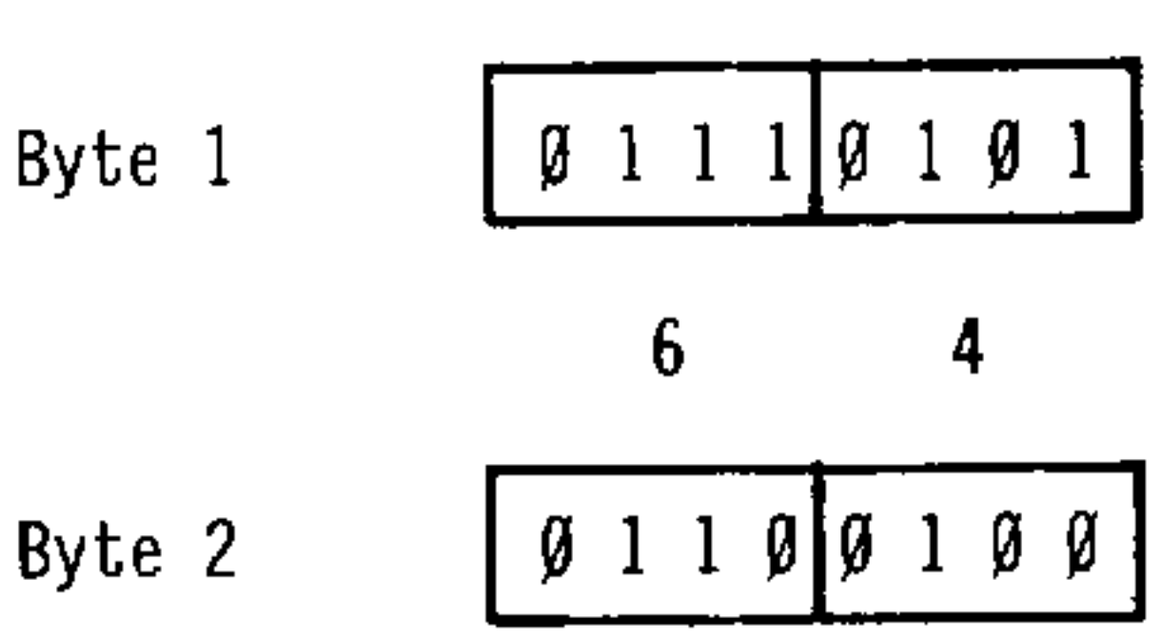
The basics of 8 and 16–bit addition and subtraction should now be understood, including the use of the carry flag.

Chapter 5

Binary Coded Decimal and Logical Operators

In addition to numbers being represented in binary, hex and decimal notation, another representation exits. This is given the rather grand name of 'Binary Coded Decimal', or BCD. Whereas in binary one number (0–255_{10}) is normally stored per byte, in BCD the byte is split into two. The name given for each 4–bit half–byte is a nybble!

For example, 7564_{10} stored in BCD requires 2 bytes (4 digits to encode at 2 digits per byte).



Note that byte 1 is not necessarily stored in memory first. Its location depends upon the storage format used in the program.

It may have crossed your mind that 4 bits are used to store each digit (0–9) when 3 bits could uniquely store them. The reason 4 bits are used is that BCD has been around much longer than micros. It was first used on mainframes where it was more efficient to store BCD digits in 4 bits. BCD is used a lot in accountancy programs as well as being very useful for passing information to certain forms of display devices (e.g. the 7–segment displays seen in calculators and digital watches).

Let's now see how two BCD numbers are added together.

Decimal	Binary
8	1000
+2	+0010
10	1010

This is fine but 1010_2 is 10 in decimal and therefore too large to be stored in one BCD nybble which is only allowed to store the numbers 0 to 9. Some form of adjustment is required for answers bigger than nine. By adding 6 to any result greater than 9 the correct result is obtained:

Thus:

First nybble	Second nybble	
0000	1010	
+0000	0110	: 6 in decimal
0001	0000	

Giving the correct representation of 10 in BCD: 0001 0000.

It would soon get very annoying if every time we performed some BCD arithmetic we had to check the validity of the result and correct it if necessary. Fortunately the Z80 contains an instruction to do this for us:

DAA	**D**ecimal **A**djust the **A**ccumulator

Now on with the programming.

Program 5.1 adds 8 and 2 together to produce the answer in BCD.

PROGRAM 5.1

```
LD A,8
ADD A,2
DAA
ADD A,65
RST 8
DEFB &9E
RET
```

Note the use of the offset (65) to bring the result into the alphabet range (Q).

The result, before the offset is added, will be the BCD representation of 10 (0001 0000). In decimal this binary value corresponds to 16. Thus the character with the ASCII code 16+65 will be printed on the screen (Q).

EXERCISE 5.1

Add 7 and 12 together in BCD then print the answer as a letter on the screen.

Answer in the solutions chapter

EXERCISE 5.2

Subtract &12 from &35 then convert the answer to BCD and print the answer as a letter on the screen.

Possible answer in the solutions chapter.

In some situations it is necessary to extract the least significant nybble from a byte. This is accomplished by removing the most significant nybble. This can be done with the instruction

AND s	**Logical AND** of operand **s**
s is any of A, B, C, D, E, H, L, (HL), (IX+d), or (IY+d).	

Anyone who has come into contact with digital electronics will recognise the following symbol. It is the symbolic representation of an AND gate.

An Elecronic AND Gate

FIGURE 5.1

The AND gate functions as follows. If and only if both inputs A and B are set (1) then the output C will be set. Describing the operation of an AND gate is fairly easy, but as the logic of gates becomes more complex it becomes increasingly difficult to describe clearly. The answer to this problem is to use a 'truth table'. A truth table enables a logic gate's output to be easily and quickly deduced for a given set of inputs.

For example here is the AND gate's truth table:

INPUTS A B	OUTPUT C
Ø Ø	Ø
Ø 1	Ø
1 Ø	Ø
1 1	1

Truth Table for Logical AND

FIGURE 5.2

Notice C is set only when both A AND B are set, hence the name 'AND'.

EXERCISE 5.3

Using Figure 5.2 decide whether C would be set or not for the following inputs

1. A=Ø B=1
2. A=1 B=1
3. A=Ø B=Ø

Answers in the solutions chapter.

When the Z80 chip performs an AND instruction it operates upon 8 bits at a time. Take a look at this:

Problem: What is the result of ANDing 10101101_2 with 00001111_2?

Solution:

```
        1Ø1Ø11Ø1
AND     ØØØØ1111
        ---------
        ØØØØ11Ø1
        ---------
```

What has happened?

The most significant nybble has been 'masked off', i.e. all the bits comprising the MSN have been set to zero whilst those in the LSN have been left untouched.

This process is very powerful in that it allows us to extract any portion of a byte using a suitable mask. e.g. To mask off the LSN of 10111011 we would AND it with 11110000:

```
        1Ø1Ø  11Ø1
AND     1111  ØØØØ

        ---------
        11Ø1  ØØØØ
        ---------
```

Generally any bit position that is required to remain intact is ANDed with one, the rest with zeros. For example, if the result of ANDing an eight–bit number with 00000011 is 2, then we know that the number, whatever it is, is divisible by 2.

EXERCISE 5.4

What mask would have to be used to result in 00000011_2 from 10101011_2?

Answer in the solutions chapter

EXERCISE 5.5

What is the result of ANDing 253_{10} with 75_{10}?

Answer in the solutions chapter.

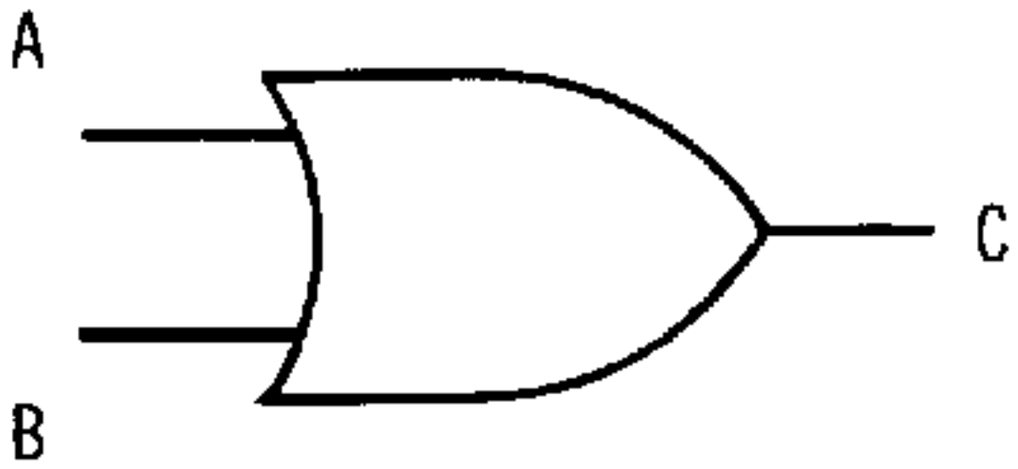
Now to write a program using the AND instruction. Program 5.2 logically ANDs 225_{10} with 254_{10} and then prints the result on the screen.

PROGRAM 5.2

```
LD C,225
LD A,254
AND C
RST 8
DEFB &9E
RET
```

This will produce the Einstein's symbol for 224 on the screen.

AND is not the only logical operator that the Z80 supports. Let's now investigate the OR gate. The standard symbol for an OR gate is



An Electronic OR Gate

FIGURE 5.3

The output C is set (1), if either A or B is set (1) or both A and B are set. This is expressed in the following truth table.

INPUTS		OUTPUT
A	B	C
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	1

Truth Table for Logical OR

FIGURE 5.4

The Z80 OR instruction is:–

OR s Logical **OR** of operand **s**

s is any of A, B, C, D, E, H, L, (HL), (IX+d), (IY+d).

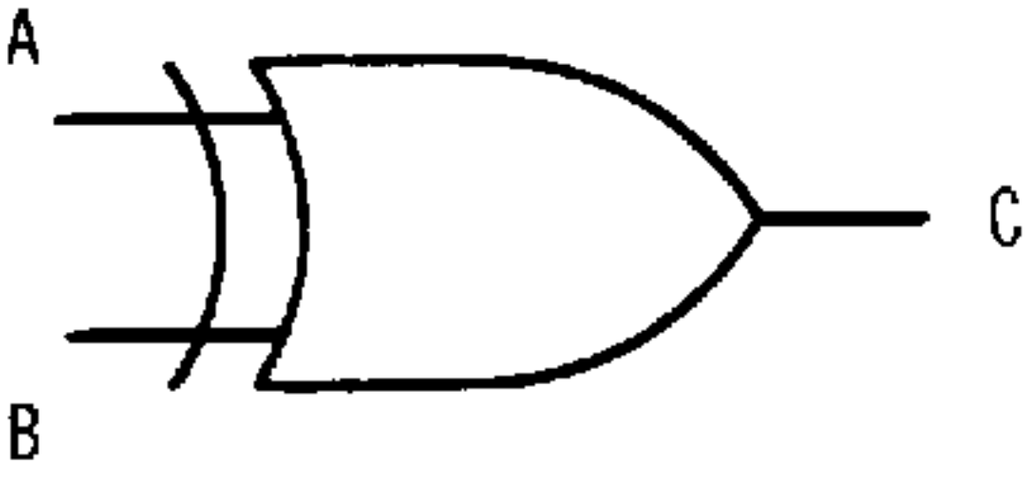
EXERCISE 5.6

What is the result of the following logical operations?

1. $1ØØ1_2$ OR $11Ø1_2$
2. $25Ø_{1Ø}$ OR $25_{1Ø}$
3. $(2Ø9_{1Ø}$ OR $2Ø_{1Ø})$ AND $27_{1Ø}$

Answers are given in the solutions chapter.

It may seem strange that an OR gate produces an output when *both* inputs are set (1); indeed this can be a problem in certain cases. Thus 'eXclusive OR' is used, overcoming this problem. The symbol used to represent an XOR (eXclusive OR) gate is:



An Electronic XOR Gate

FIGURE 5.5

INPUTS		OUTPUTS
A	B	C
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	Ø

XOR Truth Table

FIGURE 5.6

The corresponding Z80 instruction is:

XOR s eXclusive **OR** of operand **s**

s is any of A, B, C, D, E, H, L, (HL), (IX+d), (IY+d).

EXERCISE 5.7

What is the result of the following logical operations? Try them on paper first and then verify your answers by writing a program.

1. 10112 XOR 11101002
2. 7710 XOR 20019
3. (2510 OR 25510) AND 20010

Answers are given in the solutions chapter.

Signed Numbers

So far all the numbers dealt with have been positive. In many situations a need arises for negative numbers.

How can negative numbers be represented in binary? The method used allows the Z80 to treat negative numbers in much the same way as it treats positive numbers. This method is given the rather grand name 'two's complement'. Before investigating two's complement, however, it is necessary to understand the concept of one's complement!

One's Complement

When using one's complement notation all positive integers are represented in binary as usual. However, negative numbers are represented by replacing all the 1's in a byte with 0's, and replacing all the 0's with 1's.

For example:

$$+9_{10} = 1001_2$$

Now replace all the 1's by 0's and all the 0's by 1's, i.e: 1's complement +9:

1001 1's complement = 0110

Thus in 1's complement notation $-9_{10} = 0110$

But 0110 also represents $+6_{10}$; this is in fact one of the problems of 1's complement notation.

EXERCISE 5.8

Convert the following numbers to their 1's complement form.

1. 1011_2
2. 101110_2
3. 14_{10}

Answers are given in the solutions chapter.

Two's Complement

As in 1's complement, positive integers are represented normally in binary. To get two's complement, negative numbers are first 1's complemented then one is added to the result. It may seem like a strange way of representing negative numbers at first, but it works. Let's now try adding 7 and –5 using 2's complement for –5.

$5_{10} = 0101_2$

1's complement 1010

Add 1 $+0001$

$\underline{\overline{1011}} = -5_{10}$

Now:–

$7_{10} = 0111$

$+(-5)\ 1011$

$\underline{\overline{2}}\quad\underline{\overline{0010}}$ + carry

Ignoring the carry bit, the answer is correct. The fact that the carry bit can be disregarded, with the result remaining correct, considerably aids writing simple arithmetic programs dealing with negative numbers.

EXERCISE 5.9

Calculate the answer to each of the following problems using the 2's complement representation for negative numbers.

1. $-3 + 10 = ?$
2. $-1 + 7 = ?$
3. $-10 + 8 = ?$

The answers are given in the solutions chapter.

Using 4 bits, the decimal numbers represented by 2's complement notation are as below:

Decimal	2's comp.
+7	Ø111
+6	Ø11Ø
+5	Ø1Ø1
+4	Ø1ØØ
+3	ØØ11
+2	ØØ1Ø
+1	ØØØ1
Ø	ØØØØ
−1	1111
−2	111Ø
−3	11Ø1
−4	11ØØ
−5	1Ø11
−6	1Ø1Ø
−7	1ØØ1
−8	1ØØØ

The 16 unique combinations of 1's and 0's which result from 4 bits no longer represent the integers 0–15 but the integers +7 to –8. Similary when using 8 bits the valid range of numbers is +127 to –128. This can lead to problems when adding together two numbers that result in a number greater than +127.

e.g.

```
 1ØØ      Ø11ØØ1ØØ
  79      Ø1ØØ1111

179      1Ø1Ø1Ø11
```

In 2's complement 10101011 represents –85; this is clearly the wrong answer. 'Overflow' is said to have occurred. This is detected by the P/V (Parity/oVerflow) flag and it is up to the programmer how to act on this information. One blunt solution is to declare the result invalid. Overflow will generally occur under the following conditions.

When:–

1. Adding together two large positive or negative numbers.

2. Subtracting a large positive number from a large negative number or subtracting a large negative number from a large positive number.

The Z80 contains instructions which convert the contents of the accumulator into either 1's or 2's complement, as follows:

NEG	**NEG**ates the contents of the accumulator (exactly the same as 2's complement)

CPL	**ComPL**ements the contents of the accumulator (1's complement).

Program 5.3 below calculates the answer to the sum –112+104.

PROGRAM 5.3

```
        LD A,112        A=112
        NEG             A=-112
        ADD A,1Ø4       A=-112+104
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

When run, this program will print the Einstein's character for ASCII code 248 on the screen. The binary representation of 248 is 11111000 which, using 2's complement notation, represents –8 in decimal. The important point to notice here is that the Z80 cannot distinguish between 248 and –8. 2's complement is a concept the programmer uses to represent negative numbers, not the Z80, and as such, care is needed when using it to ensure the expected result is obtained.

EXERCISE 5.10

Use the CPL and INC instruction instead of NEG to calculate the result of the following sum

```
-2Ø + 98 = ?
```

A possible answer is given in the solutions chapter.

That's 'yer lot' for another chapter; now to summarise.

Summary

The following ideas and concepts should now be familiar.

1. BCD Arithmetic
2. Logical Operators
    1. AND
    2. OR
    3. XOR
    4. Logical Masks
3. Signed Numbers
    1. 1's Complement
    2. 2's Complement
    3. Overflows

The following instructions should also be recognised.

DAA	AND s
CPL	OR s
NEG	XOR s

Chapter 6

Multiplication, Division and the Rotate Group

Almost all real life computer applications require the manipulation of numbers. Whilst some only require basic adding and subtracting, many require multiplication and division. This chapter looks at how these arithmetic functions can be implemented in assembly language.

Binary Multiplication

Before we embark upon binary multiplication, let's examine the decimal multiplication process. Take the product 13x14. We define 13 as the MULTIPLICAND and 14 as the MULTIPLIER and lay out the multiplication as below:

```
  13    Multiplicand
  14    Multiplier

 ----
  52
 130
 ----

 182    Answer
```

The multiplication is performed by multiplying the multiplicand by the rightmost digit of the multiplier and storing this result as the first 'partial product', i.e. 13x4=52. Next we multiply the multiplicand by the next digit of the multiplier, resulting in the second partial product, i.e. 1x13=13. This partial product is written down shifted one position to the left. The two partial products are now added together resulting in 182, the correct answer.

It is quite possible to use the same method when performing binary multiplication.

For example, to multiply 5x7 in binary:–

5_{10} = Ø1Ø1$_2$ (working only to 4 bits)

$7_{.0}$ = Ø111$_2$

```
      Ø 1 1 1     (= 7)
    x Ø 1 Ø 1     (= 5)
    ---------
```

Adding as we go!

```
Partial Product 1       Ø 1 1 1           Ø 1 1 1
Partial Product 2     Ø Ø Ø Ø Ø           Ø 1 1 1
Partial Product 3   Ø 1 1 1 Ø Ø   1 Ø Ø Ø 1 1
Partial Product 4 Ø Ø Ø Ø Ø Ø Ø   1 Ø Ø Ø 1 1
```

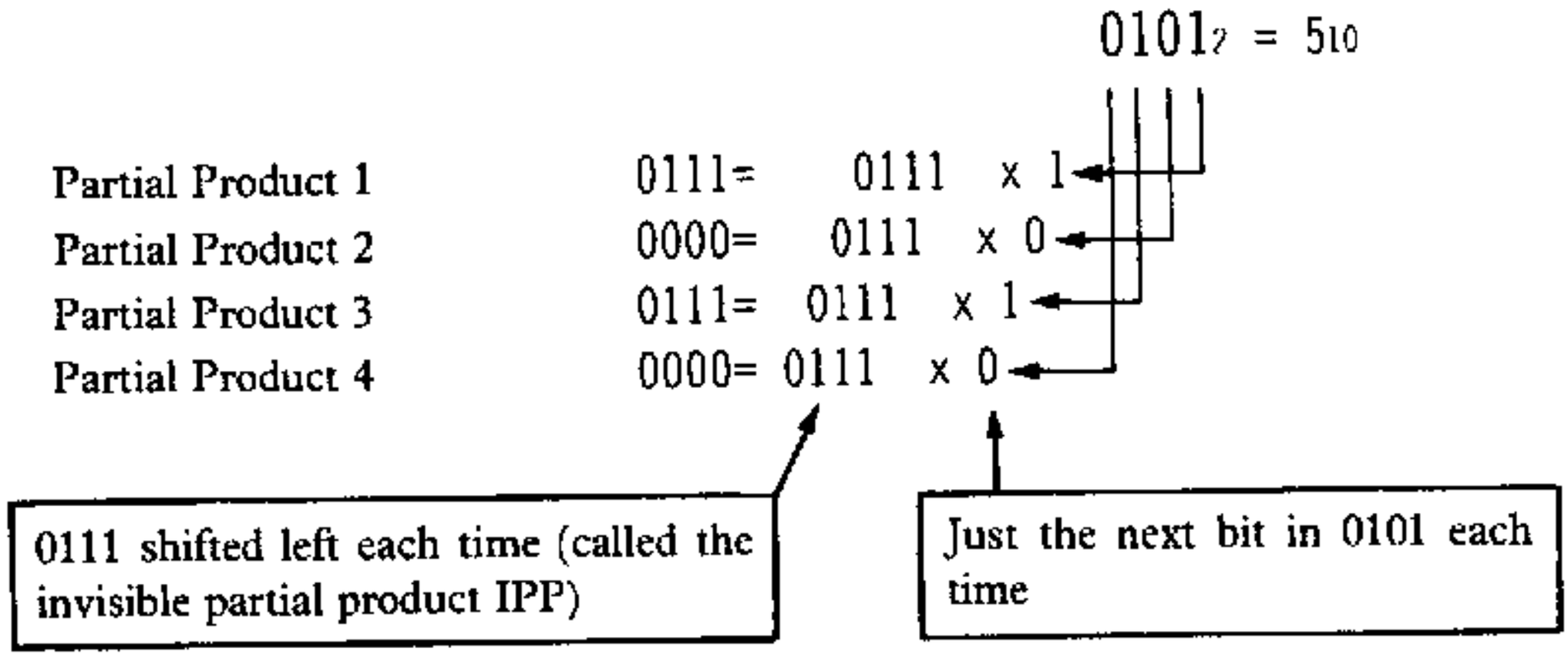
Answer 1ØØØ11$_2$

1ØØØ11$_2$= (1x32)+(Øx16)+(Øx8)+(Øx4)+(1x2)+(1x1)= 35$_{.0}$

With 1 as the rightmost digit of the multiplier, the partial product has the same pattern of digits as the multiplicand, i.e. 0111. Otherwise the partial product is zero, i.e. 0000. Every new partial product is written down shifted one position to the left. Thus binary multiplication reduces to successive additions and shifts.

8–Bit Binary Multiplication

In order to perform a multiplication using the Z80 we will use the accumulator to hold the 'running total', register C for the multiplicand and register E for the multiplier. Let's now see how the partial products are formed.


$0101_2 = 5_{10}$
Partial Product 1
Partial Product 2
Partial Product 3
Partial Product 4
0111= 0111 x 1
0000= 0111 x 0
0111= 0111 x 1
0000= 0111 x 0
0111 shifted left each time (called the invisible partial product IPP)
Just the next bit in 0101 each time

The problem of binary multiplication can be expressed in a flowchart as follows:

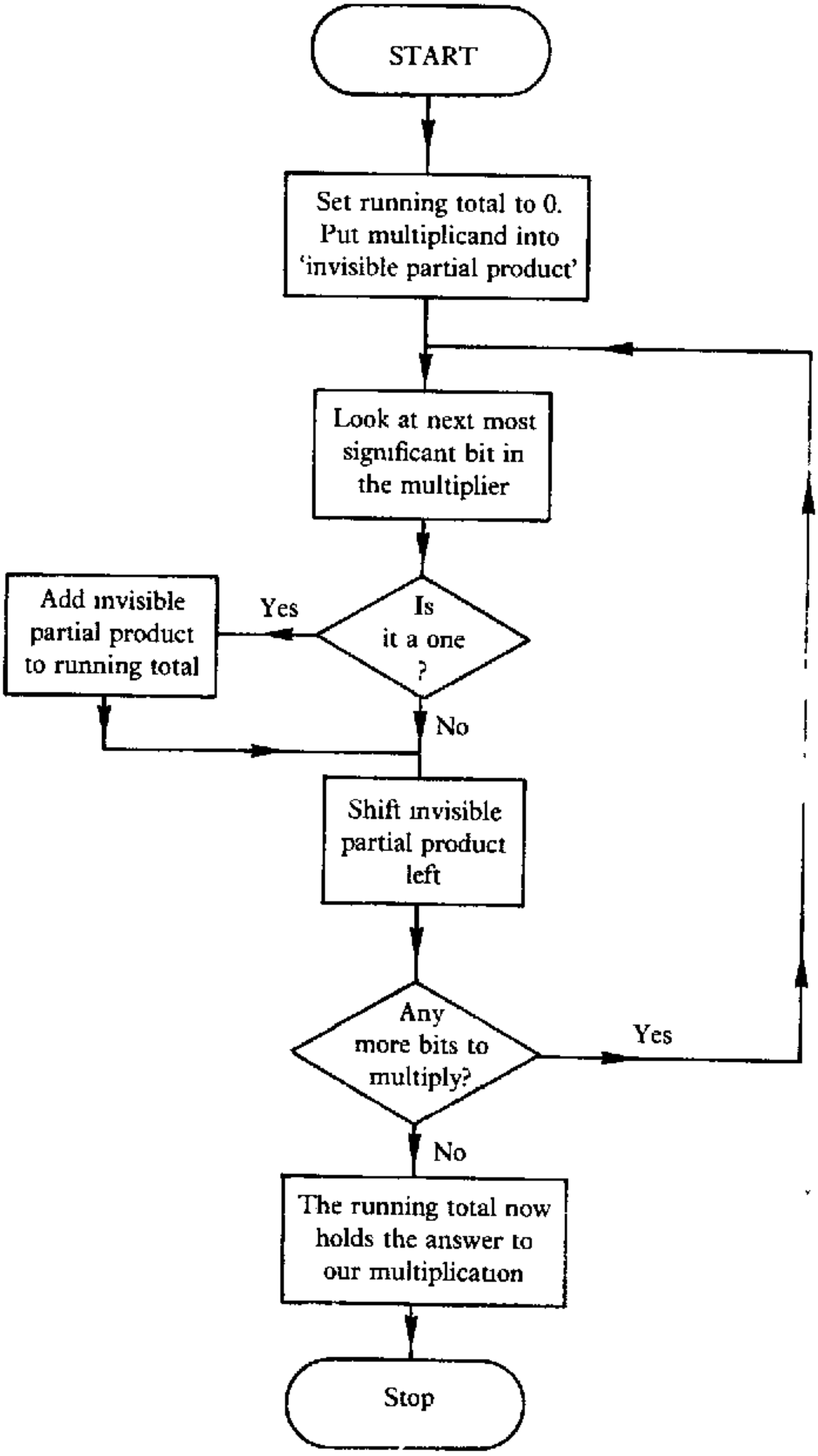
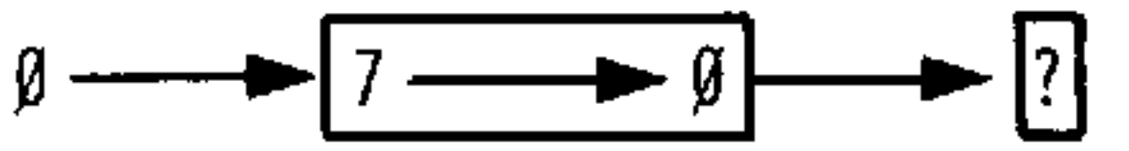

START
Set running total to 0.
Put multiplicand into
'invisible partial product'
Look at next most
significant bit in
the multiplier
Is
it a one
?
Yes
Add invisible
partial product
to running total
No
Shift invisible
partial product
left
Any
more bits to
multiply?
Yes
No
The running total now
holds the answer to
our multiplication
Stop

FIGURE 6.1

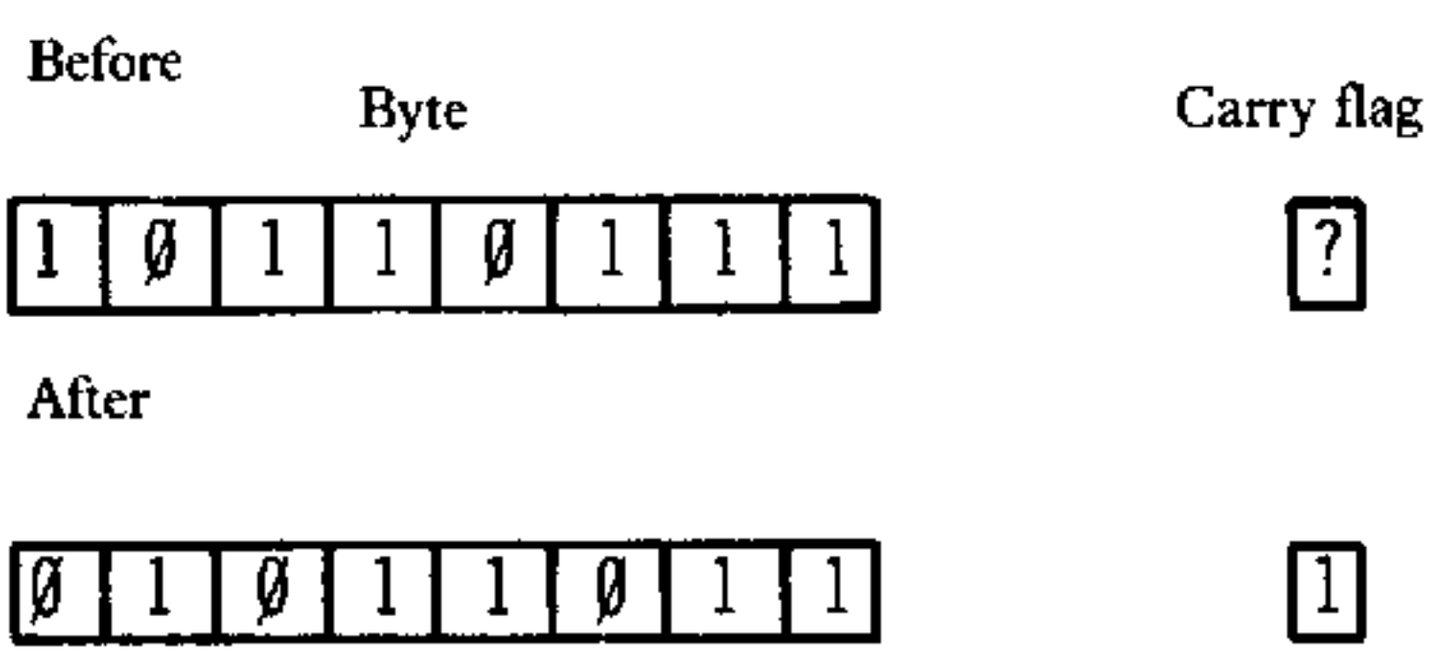
All that is required before starting to write the program are the instructions that enable us to shift bits within bytes.

SRL s	**L**ogical **S**hift **R**ight of operand **s**

Represented diagramatically:



For example, consider shifting 10110111 right with the instruction SRL.



Bit 7 has been replaced with a 0, bits 1 to 6 shifted right one position and bit 0 moved into the carry flag.

Now onto the program:

PROGRAM 6.1

```
        LD A,Ø          Zero Accumulator
        LD C,7          C = Multiplicand
        LD E,5          E = Multiplier
        LD B,4          B - Count (for 4bits)
ADD:    SRL C           Shift C right
        JR NC,NOADD:    If rightmost bit - 0 then jump to NOADD
        ADD A,E         Add IPP to running total in accumulator
NOADD:  SLA E           Shift IPP left
        DEC B           Decrement counter
        JR NZ,ADD:      Jump back to ADD if there are still some more bits to
                        multiply
        ADD A,65        Add offset
        CALL PRINT:     Print character on screen
        RET             Return
PRINT:  RST 8
        DEFB &9E
        RET
```

An offset of 65 is used resulting in the character with an ASCII code of 100 being printed on the screen, i.e. a lower case d (the answer to 7x5 was 35, 35+65=100).

EXERCISE 6.1

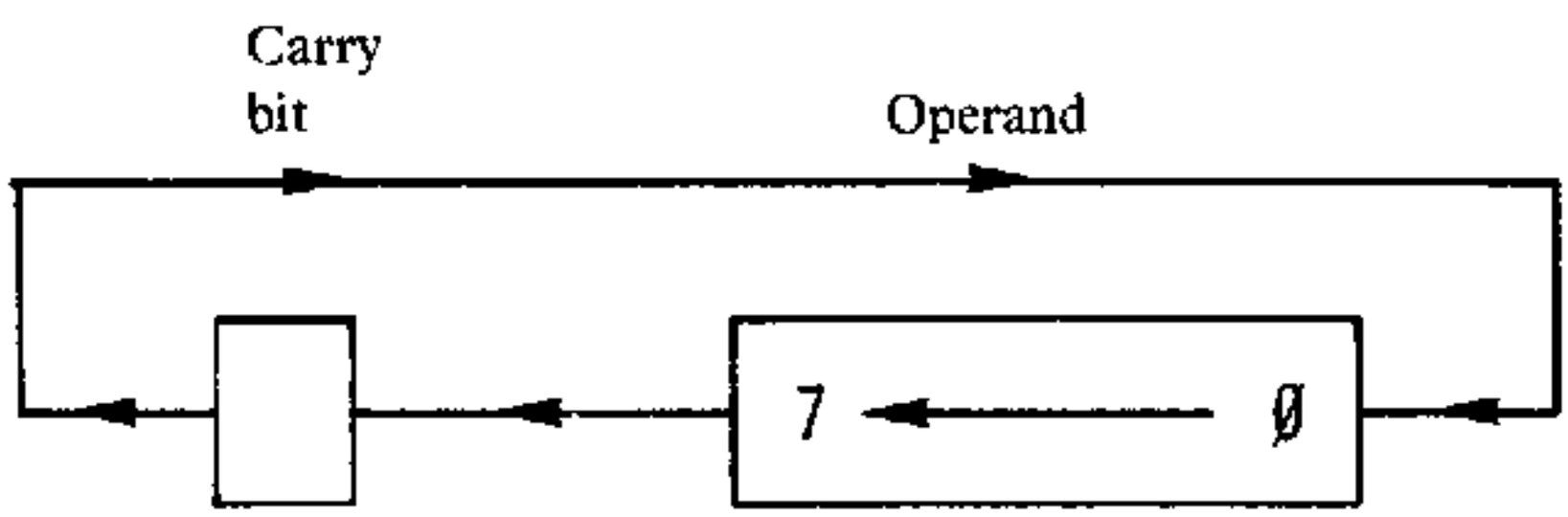
Write a program to multiply 12x5. Then print the result directly on the screen.

A possible answer is given in the solutions chapter.

Program 6.1 can only multiply together two numbers which result in a sum less than 255. The main reason for this limiting factor is that as the invisible partial product is shifted left it will eventually shift completely out of the register. A similar problem occurred when we were dealing with 16 bit addition and subtraction. The problem was overcome by re–loading the high order register with the carry bit when overflow occurred. What is now required is an instruction which, after a shift instruction on the LSB, will shift the MSB including any generated carry. This is accomplished with the following instruction:

RL s	**R**otate **L**eft operand **s** including the carry bit

Represented diagramatically:



The carry flag is loaded into bit 0, bits 1–8 are shifted left one position and the carry flag loaded with bit 7.

To shift a 16 bit register two instructions are thus required. The LSB is shifted normally using the SLA instruction, then the MSB is shifted using RL. For example, suppose we wanted to shift left the contents of DE. The following instructions would accomplish this.

```
SLA E
RL D
```

Now to incorporate this new instruction into a program.

Problem: Print the result of the following sum on the screen: 7 x 10 = ?

PROGRAM 6.2

```
        LD C,7          C – Multiplicand
        LD E,1Ø         E   Multiplier
        LD D,Ø          Zero register D
        LD B,8          B – Number of bits
        LD HL,Ø         Zero HL, it will be used to keep the running total
NXTB.   SRL C           Shift multiplicand right
        JR NC,NOADD:    If carry=0 jump to NOADD
        ADD HL,DE       Add IPP to running total
NOADD:  SLA E           Shift LSB left
        RL D            Reload carry and shift D left
        DEC B           Decrement counter
        JR NZ,NXTB:     Jump back if any more bits left
        LD A,L          Load A with answer
        CALL PRINT:     Print answer on screen ('F').
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

Note that although we used a 16 bit register for the result, due to the careful selection of the multiplier and multiplicand, only the LSB contains a value, allowing us to print it on the screen.

EXERCISE 6.2

Using Program 6.2 calculate the product 146x124. Note that this results in a 16 bit answer, thus both H and L will have to be printed on the screen.

A possible answer is given in the solutions chapter.

Program 6.2 used register B as a counter. Every time the multiplier was shifted left, B was decremented then tested. This operation requires two instructions: DEC and JR NZ. These two instructions can however be replaced by just one, making the program more efficient and elegant.

DJNZ e	**D**ecrement B and if this results in a **N**on–**Z**ero answer **J**ump to memory address **e**.

EXERCISE 6.3

Replace the DEC B and JR NZ,NXTB: with a DJNZ NXTB: instruction in Program 6.2, or your answer to Exercise 6.2

An answer is given in the solutions chapter.

Binary division is also possible using a very similar method to that used for multiplication.

Binary Division

Consider the following division:

```
10 ÷ 785
```

Firstly we try to divide 7 by 10; as 10 does not go into 7 the next step is to bring down the 8 and try to divide 78 by 10. This divides seven times with a remainder of 8.

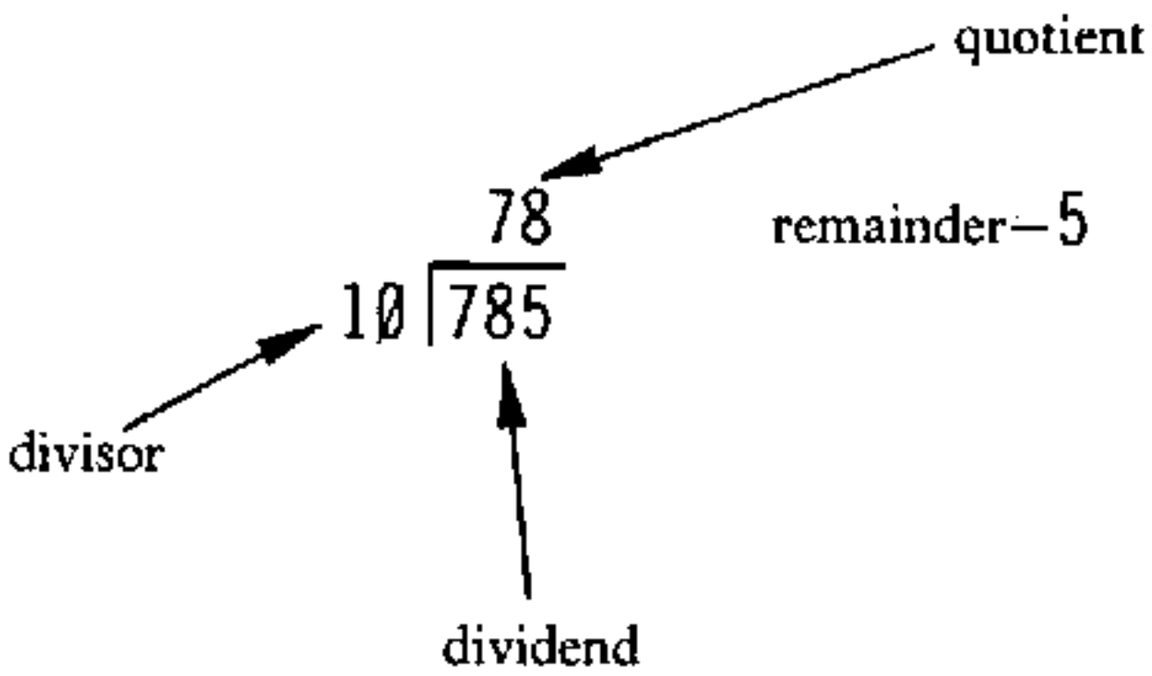
```
   7
10|785
```

As 10 'doesn't go into' 8, the next step is to bring down the 5 and try to divide 85 by 1. This divides 8 times remainder 5.

```
   78   remainder 5
10|785
```

Thus the result is 78 remainder 5. This form of division is termed 'integer division' as fractions are not considered.

As with multiplication the various numbers in a division sum have names; for example 'remainder' is already familiar. The others are as follows:



Now to consider a 16 bit by 8 bit division.

Problem: Print the result of the following division on the screen in a recognisable format.

$$2765 \div 75 = ?$$

Binary division is carried out as follows. If, without borrowing, the 8 bit divisor can be subtracted from the MSB of the 16 bit dividend, then the relevant bit in the 8 bit answer is set. This process is repeated eight times resulting in an 8 bit quotient and an 8 bit remainder.

PROGRAM 6.3

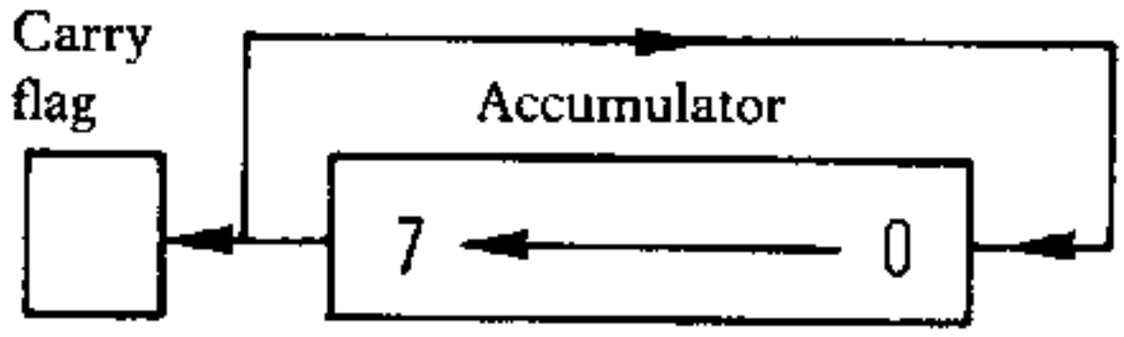
```
        LD HL,2765      HL - Dividend
        LD C,75         C  Divisor
        LD B,8          B  Count
NXT:    ADD HL,HL       Shift dividend left
        LD A,H          A – MSB of dividend
        SUB C           Subtract divisor
        JR C,NXTB:      If carry jump to NXT
        LD H,A          Reload dividends MSB
        INC L           Increment answer
NXTB:   DEC B           Decrement counter
        JR NZ,NXT:      If non–zero jump to NXT
        LD A,L          A – answer (quotient)
        CALL PRINT:     Print answer
        LD A,H          A – Remainder
        CALL PRINT:     Print remainder
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

Run the program. It will print '$A' on the screen, where the ASCII code for '$' corresponds to the quotient and 'A' the remainder. Check that these are correct. One point to note: as the Z80 doesn't contain a 16 bit left shift instruction, ADD HL,HL was used. In binary, adding a number to itself is the same as shifting it left one bit position, or multiplying by two. If you compare this with the base 10 case, say, shifting 19 left will give 190 – the same as multiplying by 10. What do you think will be the effect of shifting a binary number right one bit position? (Answer: it's the same as dividing the number by 2).

So far only a few of the Z80's shift instructions have been used in general purpose multiplication and division routines. It is sometimes easier and quicker to use a small group of shift or rotate instructions to perform a a specific arithmetic operation. By using a rotate as opposed to a shift instruction the register contents are altered, but no information is lost – i.e. no bits disappear off the end without trace. Here is an instruction which rotates the contents of the accumulator left:

RLCA	**R**otate the contents of the **A**ccumulator **L**eft and load a copy of bit 7 into the **C**arry flag.

Representing RLCA diagrammatically:



Notice that bit 7 is not lost but inserted into bit 0. An instruction to rotate the contents of the accumulator right also exists:

RRCA	**R**otate the contents of the **A**ccumulator **R**ight and load a copy of bit 0 into the **C**arry flag.

Let's now use these two instructions.

Problem: Multiply 10 by 16, print the result on the screen, divide the result by 2 and print the new result on the screen.

PROGRAM 6.4

```
        LD A,10         A=10
        RLCA            A=20
        RLCA            A=40
        RLCA            A=80
        RLCA            A=160
        CALL PRINT:     Print A ('■')
        RRCA            A=80
        CALL PRINT:     Print A ('P')
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

EXERCISE 6.4

Write a program to calculate and print the results of the following sums in the screen using the rotate and add instructions.

1. 5 x 32 = (160; a space)
2. 254 ÷ 2 = (127; a '■')

Possible answers are given in the solutions chapter.

The Z80 supports more rotate and shift instructions than have been used; these are detailed in the appendices under 'The Z80 Instruction Set'.

Now let's have a look at a set of instructions that allow individual bits within a byte to be set, reset or tested.

The Bit Set, Reset and Test Group

Consider the following instruction:

BIT b,r	Tests the **BIT** at position **b** in the operand **r.**

This instruction is used whenever a particular bit in a byte needs to be tested e.g: in input/output operations.

To illustrate the use of this command, a register will be zeroed, incremented until bit 7 is set and the resultant register contents will then be printed on the screen.

PROGRAM 6.5

```
        LD A,Ø          A -0
NXT:    INC A           Increment A
        BIT 7,A         Test bit 7 of accumulator
        JR Z,NXT:       Is bit 7 set?
        CALL PRINT:     Print contents of A on the screen (a space)
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

This program loads A with zero and repeatedly increments it until, eventually, bit 7 is set. Run it; no great effect is noticed as bit 7 being set corresponds to 128 in decimal, the ASCII for a space. Try inserting either an INC A or a DEC A instruction just before the CALL PRINT: instruction to print something visible on the screen.

Two further bit instructions exist: one to set and the other to reset an individual bit within a byte.

SET b,r	**SET**s the bit at position **b** in the operand **r**.

RES b,r	**RES**ets the bit at position **b** in the operand **r**.

To illustrate these instructions, consider the ASCII code for 'C', in binary 01000011. If bit 0 is reset, the effect is the same as subtracting one, resulting in the ASCII representation of 'B'.

PROGRAM 6.6

```
LD A,67          A  ASCII for 'C'
CALL PRINT:      Print 'C'
RES Ø,A          Reset bit 0
CALL PRINT:      Print 'B'
SET Ø,A          Set bit 0
CALL PRINT:      Print 'C'
RET
```

EXERCISE 6.5

Write a program that loads the accumulator with 255, prints the contents on the screen, resets bit 4, prints the result on the screen, sets bit 4, resets bit 3 and then prints the final result on the screen.

A possible answer is given in the solutions chapter.

Well, that concludes another chapter; by now you should have a fair idea as to how to write a fairly complex program. Now to summarise:

Summary

The following ideas and instruction groups should now be reasonably familiar:

1. Binary multiplication and division
2. The shift and rotate group.
3. The bit set, reset and test group.

Chapter 7

The Stack

The 'stack' is a block of memory. It is used for the rapid transfer of data, and is filled downwards, the next vacant location being recorded by a register called the 'Stack Pointer', or SP. The usual analogy is with a stack of plates, only the top one being accessible as this was the last one put there. This mode of filling and emptying the stack is known as 'last in, first out' or LIFO, so the stack is a LIFO store.

One function of the stack is to record addresses during subroutine calls, which the Z80 does automatically. When the Z80 sees an instruction such as CALL PRINT:, it must first of all record where the next instruction in the calling routine (main program) is, so that it can find it again after the subroutine has been executed. It then places the CALL ADDRESS into the PC (program counter).

Suppose the Z80 is about to execute the following instruction:

CALL PRINT. where PRINT. points to a location in memory.

The Z80 does the following things:

1. Increment PC
2. Fetch command byte ('CALL')
3. Increment PC
4. Fetch first byte of operand (1st byte of CALL ADDRESS)
5. Increment PC

6. Fetch second byte of operand (2nd byte of CALL ADDRESS)

7. Store MSB of PC on stack

8. SP – SP–1

9. Store LSB of PC on stack

10. SP – SP–1

11. Put CALL ADDRESS into the PC

12. Execute subroutine, up to RET

13. Increment PC

14. Fetch command byte ('RET')

15. Load LSB of PC from stack

16. SP – SP+1

17. Load MSB of PC from stack

18. SP = SP+1

19. Increment PC

20. Continue with main program

In this example, the subroutine could well have met further subroutines or 'nested' subroutines, and each time a CALL was executed the return address would have been piled onto the stack. Then as the program RETurned successfully through these subroutines, the return addresses would have been stripped off successively to steer the Z80 back to the original point of departure. The area of memory used for the stack is more than 256 bytes long, so there should be plenty of room.

Fortunately, the operation of the stack in recording addresses when executing subroutines is automatic and so the programmer can allow the Z80 to do the job. However, when using built–in subroutines, the stack does not automatically store register contents but must be programmed to do so. The instructions for doing this are:

PUSH and POP

PUSH qq **PUSH** register pair **qq** onto the stack

qq is any of AF, BC, DE and HL.

PUSH also comes in a form suitable for storing IX and IY:

PUSH IX **PUSH** the **IX** register onto the stack

PUSH IY **PUSH** the **IY** register onto the stack

These instructions copy the contents of the specified registers onto the stack. Having dealt with the subroutine or whatever, to restore the former contents of the registers, the instruction POP is used:

POP qq Retrieve register contents from the stack

qq is any of AF, BC, DE or HL.

POP IX	**Retrieve IX** from the stack

POP IY	Retrieve **IY** from the stack

Whether you are using PUSH or POP, there is no need for you to update the stack pointer – this is taken care of automatically.

When storing register contents on the stack, it is important to remember that the stack is a LIFO structure: you will normally retrieve the registers in the reverse order to that in which you stored them. Also, when writing a program incorporating PUSH and POP, every PUSH should be matched by a POP, otherwise the computer could get mixed up: it might, for instance, think that a number in the stack is a RETurn address, when in fact it is the unPOPed BC register pair. Obviously, this could lead to a system crash. Program 7.1 is an example of using PUSH and POP correctly, to preserve A and F (F is the Flags register).

It uses two subroutines to perform the simple task of printing characters to the screen (in this case 'M') in diagonally adjacent positions. Two 'special' locations &FB4A (x–cursor) and &FB4B (y cursor) are loaded with the x and y coordinates, and these in turn are used to print at the required place.

PROGRAM 7.1

```
        LD A,&4D        LOAD 'A' with ASCII character 'M'.
        LD D,6          Set x position & store in D register.
        LD E,4          Set y position & store in E register.
        PUSH AF         Copy A onto STACK.
        CALL CURSET:    Call the cursor-setting subroutine.
        POP AF          It is now safe to recall A from STACK.
        CALL PRINT:     Print the contents of the 'A' register.
        INC D           Increase the x-coordinate.
        INC E           Increase the y-coordinate.
        PUSH AF         Copy A to STACK again (preserve it).
        CALL CURSET:    Re-position cursor at new location.
        POP AF          Get the 'M' back into Accumulator.
        CALL PRINT:
        RET             Back to DOS.
CURSET: LD HL,&FB4A     Load HL with address to hold x-position.
        LD (HL),D       Now load &FB4A with contents of D.
        LD HL,&FB4B     Load HL with address to hold y-position.
        LD (HL),E       Now load &FB4B with contents of E.
        RET             Return from CURSET routine.
PRINT:  RST 8
        DEFB &9E
        RET             Return the PRINT routine.
```

This will produce two 'M's on the screen in consecutive diagonal locations. Note that the contents of the accumulator were saved to the stack by a PUSH instruction immediately prior to calling the CURSET subroutine; they were then RECALLED from the stack using POP so that we had an 'M' ready in A for the PRINT: routine to use. This is just done to provide an example, but it is often necessary because many of the various built-in routines need to use one or more of the registers themselves. Another point to note is that although we were only interested in the A register, we had to PUSH and POP the AF pair. This is because the PUSH and POP instructions only work on register pairs.

EXERCISE 7.1

Write a program to store the following ASCII characters in the registers shown.

Register	ASCII character
B	"B"
C	"C"
D	"D"
E	"E"

After storing the characters, print them to the screen, then swap the contents of the register pairs BC and DE. This can be done simply by PUSHing them onto the stack and then POPing them off again in reverse order (i.e. PUSH BC; PUSH DE;...POP BC; POP DE: this may not look like reverse order, but remember that stack operations take place in LIFO – Last In, First Out). To prove that the register contents have been swapped, print them out again.

A solution is in Solution Chapter.

The Stack Pointer (SP)

To keep track of which items to POP next, or where to place the next item to be PUSHed onto the stack, the Z80 uses a special register called the Stack Pointer, or SP. This normally points to the last location in the stack – i.e. to the last byte within the stack.

There are several instructions that allow the programmer to access the stack pointer, and change it. Using these, it is possible to set up a 'user stack'. After adjusting SP to point to some new memory location, the Z80 will use the new location as the start of a new stack, completely ignoring the old one. Thus, if a new stack is set up within a subroutine, it is important to preserve any RETurn addresses by PUSHing them onto the new stack.

There are three LD instructions which can be used to alter SP:

LD SP,HL	**LoaD SP** with the contents of **HL**

LD SP,IX	**LoaD SP** with the contents of **IX**

LD SP,IY	**LoaD SP** with the contents of **IY**

To find out the current value of SP, the following instruction can be used:

LD (nn),dd	**LoaD** memory location **nn** with the contents of register pair **dd**

dd can be any of BC, DE, HL or in this case, SP.

Just to prove that setting up your own stack will work, try the following program:

PROGRAM 7.2

```
LD A,43          Put '+' in A.
PUSH AF          Store A in current stack.
CALL PRINT:      Put '+' on screen.
LD (&714A),SP    Remember current value of SP.
LD HL,&7148      Get ready to:
LD SP,HL         Set up new stack at &7148.
LD A,61          Put '=' in A.
PUSH AF          Store A in new stack.
CALL PRINT:      Put '=' on screen.
LD HL,(&714A)    Find original location of SP.
LD SP,HL         Go back to original stack.
POP AF           Retrieve '+'.
CALL PRINT:      Put '+' on screen.
RET
```

Program 7.2 will put '+=+' on the screen. If the stack pointer had not been moved to a new location, then the POP AF would have retrieved an '=' instead, and the screen would have displayed '+=='. Just to prove that there is a new stack at &7148, try the following short exercise!

EXERCISE 7.2

Extend the above program so that it retreives the '=' and displays it and then puts SP back where it came from again to retrieve and display another '+'. The screen should then display '+=+=+'.

Hint: Remember that SP should point to the last location within the stack, so moving it to &7148 will not work. Use &7146.

A solution is given in the solutions chapter.

To conclude this chapter, it is worth mentioning the other commands that can affect the stack. These allow the advanced (and careful!) programmer to mess about with the contents of the stack without necessarily PUSHing or POPing repeatedly to get the SP to point to the desired element in the stack.

First of all, there are three more EXchange instructions:

EX (SP),HL	**EX**change contents of **HL** with the top of the stack.

EX (SP),IX	**EX**change contents of **IX** with the top of the stack.

EX (SP),IY	**EX**change contents of **IY** with the top of the stack.

The other instructions which can affect SP are:

INC ss	**INC**rement register pair **ss**

DEC ss	**DEC**rement register pair **ss**

ss is any of BC, DE, HL or SP.

Note that these forms of INC and DEC do not affect the carry flag.

ADD IX,pp **ADD** contents of register pair **pp** to **IX**

Where pp is any of BC, DE, IX or SP.

ADD IY,rr **ADD** contents of register pair **rr** to **IY**

Where rr is any of BC, DE, IY or SP.

Summary

Having read this chapter, you should know about the following:

LIFO
SP
PUSH
POP
EX (SP),HL
EX DE,HL
Which part of the stack SP points to
User stacks
LD (nn),dd
INC ss
DEC ss
ADD IX,pp
ADD IY,rr

Chapter 8

Block Moves and Compares

The Z80 has several instructions designed to allow whole sections of memory to be handled easily, and without the program having to specifically include the addresses of each individual memory location. These instructions can be divided into two classes. The block moves allow areas of memory to be copied from one place to another. The block compares are used to search a region of memory for some specific data item.

Block Moves

The first of the block move instructions to be looked at in this chapter is LDI:

LDI	**LoaD** memory from memory and **I**ncrement data pointers

To use this instruction, HL should contain the address of the memory location that the data is to come from, and DE should contain the address of the memory location that the data is to be copied into. After executing this instruction, the Z80 increments both DE and HL. Also it decrements BC; this makes LDI particularly useful in loops – BC can be used as a loop counter. When BC is decremented to 1, the parity/overflow flag is reset to 0 – at other values of BC, LDI sets this flag to 1, so that it can be tested to end a loop upon being set, using the PO condition. This is described in more detail below; but first the program.

As an example of using LDI, Program 8.1 fills an area of memory with ASCII "H" characters and then copies them from &5000 to &6000. To prove that this happens, both memory ranges are then printed to the screen, with the codes for CARRIAGE RETURN (0A) and LINE FEED (0D) inserted between each block to separate them on the screen.

PROGRAM 8.1

```
        LD HL,&5000      Set start address for data.
        LD BC,26         Set a counter (decimal 26).
FILL:   LD A,"H"         Store "H" in A register.
        LD (HL),A        Store "H" at memory location &5000.
        INC HL
        DEC BC
        LD A,B           Load B reg into A reg.
        OR C             Test to see if both B & C are zero.
        JP NZ,FILL:      If BC>0 then jump back (keep loading
                         successive memory locations with ASCII "H".
        CALL RESET.      Re-set HL,DE,BC
LOOP:   LDI              Copy a byte of data from &5000 to &6000.
        JP PO,FINISH:    Exit loop if BC=1.
        JP LOOP:         Carry on copying if BC>1.
FINISH: CALL RESET:      Re-set HL,DE,BC again.
BLOK1:  LD A,(HL)        Load ASCII "H" from &5000 into A reg.
        CALL PRINT:      Print it to screen.
        INC HL           Move text pointer up 1 place.
        DEC BC           Decrement the count (26)
        LD A,B
        OR C
        JP NZ,BLOK1:     Keep printing if BC not zero.
        LD A,&A          Now send line feed to screen.
        CALL PRINT:      Send it to screen
        LD A,&D          Load code for carriage return.
        CALL PRINT:
        CALL RESET:
BLOK2:  LD A,(DE)        This is the same as BLOK1 except that we
                         are using the DE registers to point to text.
```

```
        CALL PRINT:
        INC DE
        DEC BC
        LD A,B
        OR C
        JP NZ,BLOK2:
        RET                     Finished so back to DOS.
RESET:  LD HL,&5000             Start of BLOK1.
        LD DE,&6000             Start of BLOK2.
        LD BC,26                How many "H"'s shall we print?
        RET                     Return from RESET subroutine
PRINT:  RST 8
        DEFB &9E
        RET                     Return from PRINT subroutine
```

Parity

As stated earlier, the parity/overflow flag is set when it is decremented to 1 rather than zero, so it is necessary for accurate memory copying to load BC with the number of bytes to be copied, plus 1. To test the parity/overflow flag, the conditions are PO – 'parity odd' and PE – 'parity even'.

'Parity' refers to the number of bits set in the byte or flag being tested: even parity means that an even number of bits are set and odd parity means that an odd number of bits are set. Thus, when the parity flag is set, it contains an odd number (1) of set bits, and can be tested with the PO condition, as in Program 8.1 above:

```
JP PO,FINISH:
```

Should you ever need to test the parity of a byte of data, this can be done by loading A with 255 and ANDing it with the byte in question. The parity flag will be set appropriately and can be tested with a statement such as the conditional JP above. Parity tests are usually used in communications systems: one bit of each byte will be set aside for use as a 'parity bit' which will be set or reset appropriately for each byte so that all bytes transmitted and received will have identical parity. If any interference garbles the data, the chances are that the parity of some bytes will have changed, and this can be detected.

Program 8.1 illustrates, indirectly, the main purpose of the LDI block move instruction: you can put extra instructions in between successive operations of the LDI instruction. Program 8.1 also illustrates LDI's main drawback: if you don't require any extra instructions between each operation of LDI, you still have to have a jump back to the LDI to get it to repeat. This wastes both memory space and time. The Z80 has a fix for this:

LDIR	**LoaD** memory from memory, **I**ncrement data pointers and **R**epeat

This instruction acts in exactly the same way as LDI except that it automatically repeats itself until BC has been decremented to 0. Also, it sets the parity/overflow flag to 0 regardless of the current value of BC. This doesn't matter of course, as with automatic repetition there is no need to do any tests to find out whether it has finished or not anyway. Program 8.2 is a re-written version of Program 8.1, replacing LDI, JP PO,FINISH and JP LOOP with LDIR.

PROGRAM 8.2

```
        :
        :
        :
LOOP:   LDIR
FINISH. CALL RESET:
BLOK1:  LD A,(HL)
        CALL PRINT:
        :
        :
        :
```

Note that in this case, it is not necessary to load BC with the number of bytes to be transferred plus one, as LDIR stops when BC is zero, whereas in Program 8.1 the loop was ended when BC=1. BC need only be loaded with the actual number of bytes to be transferred.

There are two more block move instructions:

LDD	**LoaD** memory from memory, **D**ecrement data pointers

LDDR	**LoaD** memory from memory, **D**ecrement data pointers and **R**epeat

These are the same as LDI and LDIR respectively except that DE and HL are decremented rather than incremented.

Compares

It is often handy in a program to be able to compare two values, and then to make a decision according to the result – as in the BASIC IF...THEN construction. The assembly–language equivalent is

CP s	**ComP**are **s** with A by subtracting s from A, leaving A unchanged

's' is any one of the following: A, B, C, D, E, H, L, (IX+d), (IY+d), (HL).

After subtracting s from A, the result is not kept (i.e. A is not corrupted) although the consequences of the subtraction are recorded by the carry, zero, overflow, sign and half–carry flags. The overflow, sign and half–carry flags are discussed elsewhere in the book. They are used when you wish to compare two's complement numbers.

Since the comparison subtracts s from A, the carry flag will be set if s>A, and reset otherwise. If s=A then the zero flag will be set, otherwise reset. Program 8.3 demonstrates the use of CP.

PROGRAM 8.3

```
        LD A,&21          Put a number into A.
        LD B,&4Ø          Put a number into B.
        CP B              Compare B with A.
        JP C,BIGGER:      If carry set, B>A.
        JP Z,EQUAL:       If zero set, B=A.
        LD A,&73          Else B<A;Put ASCII 's' in A.
        CALL PRINT:       Put on screen.
        LD A,&3C          ASCII '<'.
        CALL PRINT:       Put on screen.
        LD A,&41          ASCII 'A'.
        CALL PRINT:       Put on screen.
        RET               Exit program.
BIGGER: LD A,&73          ASCII 's'.
        CALL PRINT:       Put on screen.
        LD A,&3E          ASCII '>'.
        CALL PRINT:       Put on screen.
        LD A,&41          ASCII 'A'.
        CALL PRINT:       Put on screen.
        RET               Exit program.
EQUAL:  LD A,&73          ASCII 's'.
```

```
        CALL PRINT:          Put on screen.
        LD A,&3D             ASCII '—'.
        CALL PRINT:          Put on screen.
        LD A,&41             ASCII 'A'.
        CALL PRINT:          Put on screen.
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

Try this program out again with values &40 in A and &21 in B, and with &21 in both A and B to check that it works as you might expect it to.

Not only does the Z80 have the above compare instruction, it has some block compare instructions as well. The first of these is CPI

CPI	**ComPare** A with memory, **I**ncrement data pointer.

As with LDI, this instruction decrements BC, the 'Byte Counter'. Also, the parity flag is set when BC is decremented to 1, otherwise it is reset. This instruction does not use the carry flag, however, but it does affect the zero flag. This means that CPI can only be used to test for equality, not greater than or less than. Program 8.4 searches memory for a bracket '(' and puts one '(' on the screen for each one it finds in memory.

PROGRAM 8.4

```
        LD A,&28        Put '(' in A.
        LD HL,Ø         Start at beginning of memory.
        LD BC,Ø         Search 64k (BC=bytes+1,0–1—FFFF).
LOOP:   CPI
        JP PO,FINISH:   Finish if parity flag set.
        CALL Z,FOUND:   Call 'FOUND' if zero set.
        JP LOOP:        Loop again.
FINISH: CALL Z,FOUND:   Check zero flag before finishing.
        RET             Back to the assembler.
FOUND:  CALL PRINT:     '(' on screen.
        RET             RETurn from subroutine.
PRINT:  RST 8
        DEFB &9E
        RET
```

One thing to note in this program is that the 'FOUND' subroutine needs to be called from within 'FINISH' because if Z was tested before PO, and a '(' was found, calling PRINT: would corrupt the parity flag. If PO is tested first this problem does not arise.

Another block compare instruction is:

CPIR	**C**ompare A with memory, **I**ncrement data pointer and **R**epeat

This instruction is the same as CPI except that it automatically continues either until a match is found, or until BC=0. Program 8.4 can be re–written simply by changing CPIR for CPI. The new program will run faster with CPIR because the conditions do not need to be tested after every comparison.

The last two block comparisons are the same as CPI and CPIR except the data pointer, HL, is decremented rather than incremented:

CPD	**ComPare A with memory, Decrement data pointer**

CPDR	**ComPare A with memory, Decrement data pointer and Repeat**

Unlike LDD and LDDR, these instructions do still effect the parity flag.

In case you ever need to know, a table of the effects of comparisons on the testable flags C, S and V is to be found in the appendices. This should be referred to if you ever need to compare two's complement numbers (i.e. numbers greater than 128).

Summary

Having read this chapter, you should now be aware of:

LDI
PE
PO
parity
a delay loop
LDIR
LDD
LDDR
CP s
CPIR
CPD
CPDR

Chapter 9

Special Operations and Interrupts

This chapter is about some of the least useful features of Z80 assembly language, at least from a purely software point of view. Many of the special operations and features described in this chapter are hardly used at all by most programmers. However, it is worth being aware of them. Interrupts, discussed below, are very useful, but detailed discussion of them is beyond the scope of this book as they are really a matter for a hardware manual.

Interrupts

When doing a job that has to be done, no–one likes interruptions until the job is finished. The Z80 is like that too. Whilst executing a piece of program, it has all its registers under control, and all its flags set appropriately.

An interrupt, however, is a subroutine that demands execution when IT is ready, and not the Z80. In other words, an interrupt comes from outside the Z80's direct field of control – either from some external device, or from the keyboard. Thus, flags have to be stored by the Z80 – usually on the stack. Then the Z80 will 'service' the interrupt – i.e. do whatever it is required to do by the interrupt, and then it must restore all the registers and flags and continue with the original program.

Interrupt handling should be allowed for in any program which expects interrupts, particularly if the program does certain jobs which must not be interrupted. If, for instance, another device is sending a stream of data into memory, then a 'hand–shaking' procedure is carried out between the two machines. Quite simply, this is an exchange of messages like: "I am ready to send data, are you ready to receive it?" "Yes." "Here's the data ..... end of data." "Thanks!"

If such an exchange is interrupted, then the data is likely to become garbled, and hence worthless. During such periods when no interrupting is allowable, the program can block most interrupts – not all – to allow a particular process to be completed. Interrupts which can be blocked are called 'maskable' interrupts, and interrupts which cannot be blocked are called 'non–maskable' interrupts, or NMI's.

The instruction to block all maskable interrupts is DI:

DI	**D**isable maskable **I**nterrupts

Having completed the, preferably short, section of the program which cannot be interrupted, interrupts can be re–enabled using EI.

EI	**E**nable maskable **I**nterrupts

When the Z80 services an interrupt, it does this by means of a machine–code subroutine. In common with other subroutines, they must be terminated with a return statement. For non–maskable interrupts the instruction is:

RETN	**RET**urn from **N**on–maskable interrupt

For maskable interrupts it is:

RETI	**RET**urn from **I**nterrupt

If you think about the above, you may realise that interrupts could be interrupted. There are priorities, however: for instance, a maskable interrupt will not normally be allowed to interrupt a non–maskable interrupt – it will usually just have to wait.

The highest priority interrupt of all is a 'bus request' or BUSRQ. With other maskable and with non–maskable interrupts, the Z80 will at least finish executing the current instruction before dealing with the interrupt. With a BUSRQ, however, the Z80 reacts on the next 'tick' of its internal clock – i.e. on its next cycle – whether it has finished the current instruction or not.

How the Z80 reacts to maskable interrupts depends on the current 'interrupt mode'. In all cases, it first of all disables further (maskable) interrupts, and saves the PC on the stack. It is necessary always to re–enable interrupts (EI) before returning from your servicing routines.

Interupt Modes For Maskable Interrupts

The default mode is mode 0. It can be set within a program using the instruction:

IM 0	Set **I**nterrupt **M**ode **0**

Upon receiving and accepting a maskable interrupt in mode 0, the Z80 will expect the external device to give it an instruction on its next clock–cycle via its general input line, the 'data bus'. This instruction is usually a CALL to an interrupt servicing routine that the programmer has placed somewhere in memory. It is also often a ReSTart instruction:

RST n	**ReST**art at address **n**

The address n is a single byte address and can only be one of the following: &0, &8, &10, &18, &20, &28, &30 and &38. Because RST is a single byte instruction, it is often used when a fast response to the interrupt is required. The problem with it is that there is only really room at those memory locations for a jump to some other address – especially if there are several interrupt servicing routines, each required by differing devices using different RST addresses.

Both CALL and RST automatically cause the PC to be saved onto the stack. The Z80 disables further maskable interrupts and begins the servicing routine. If the programmer requires the registers and flags to be preserved, then the interrupt servicing routine should begin with something like this:

PROGRAM 9.1

```
PUSH AF
PUSH BC
PUSH DE
PUSH HL
PUSH IX
PUSH IY
```

It should end with something like:

PROGRAM 9.2

```
POP IY
POP IX
POP HL
POP DE
POP BC
POP AF
EI
RETI
```

Notice the 'AF'. This is the accumulator and the flags.

The next interrupt mode is mode 1:

IM 1	set **I**nterrupt **M**ode **1**

This mode is similar to interrupt mode 0 except that the Z80 automatically executes a RST &38 upon accepting a maskable interrupt. Also, the Z80 will ignore the contents of its data bus in the clock cycle following the interrupt. The final mode is mode 2:

IM 2	set **I**nterrupt **M**ode **2**

In this mode, after finishing the current instruction, saving the PC and disabling further maskable interupts, the Z80 will jump to any specified even–numbered memory location – i.e. one in which the least significant bit is a 0. The most significant eight bits of the address should be set in advance in a special register called the 'interrupt vector' register, or 'I' register. The least significant eight bits of the address are supplied by the interrupting device. This allows the Z80 to jump to any one of up to 128 addresses held in a table in memory starting at the high–byte address held in the I register.

The Alternate Registers

You are by now well aware that the Z80 has many different registers, in particular, A, B, C, D, E, H, L and the flags. Well, the Z80 has a second set of these particular registers, known as he 'alternate register set': A', B', C', D', E', H', L' and the alternate flags F'.

Although they can be used by the programmer, it is not advisable on the Einstein. Using the built–in routines will corrupt them, as will any interrupt. The instructions to use them are as follows:

EXX	**EX**change register pairs

This instruction swaps the contents of the register pairs BC, DE and HL with their alternates. Note that this means that any following instruction using say DE will now use the new contents of DE, that used to be in DE'.

If the alternate registers B' or C' are used, they must be restored prior to swapping back to the original register set and enabling interrupts. Thus a program to use the alternate registers might look like this:

PROGRAM 9.3

```
DI          Disable interrupts.
EXX         Exchange register pairs.
PUSH BC     Save new BC.
      .
      .
      .
POP BC      Restore BC.
EXX         Back to original registers.
EI          Enable interrupts.
```

Any such program should be kept short, as various automatic operating-system interrupts such as timers, keyboard scanning etc. need to use the alternate registers, and anything other than a short routine could cause problems. This is true whenever interrupts are disabled.

Another instruction allows A' and F' to be used:

EX AF,AF' **EX**change contents of **AF** and **AF'**

F' must be restored prior to 'unexchanging'. Again, interrupts must be disabled whilst using these registers.

Note that exactly the same instruction is used to unexchange – i.e. EX AF,AF', not EX AF',AF.

More EXchange Instructions

There are two other forms of EX which are worth mentioning:

EX DE,HL	**EX**change the contents of **DE** with the contents of **HL**

and

EX (SP),HL	**EX**change the top of the stack with the contents of **HL**

EX (SP),IX	**EX**change the top of the stack with the contents of **IX**

EX (SP),IY	**EX**change the top of the stack with the contents of **IY**

The top byte on the stack goes into L and vice–versa, and the next byte on the stack goes into H (and vice–versa). Similarly with IX and IY.

HALT

HALT	**Halt** program execution

This instruction stops the Z80 until an interrupt is received. The only thing it does is keep 'refreshing' the memory. As you know, if you switch the computer off, it forgets any program or data which is held in its memory. This is because the memory consists of 'dynamic RAM' or 'Random Access Memory'. If this memory isn't continually updated or 'refreshed' – i.e. if the internal voltages are not continually corrected, then these voltages die away and the data stored disappears with them. The Z80 uses a 'refresh register' to tell it which memory bank to refresh at any one moment.

The INs and OUTs of the Z80

The Z80 has a family of instructions designed to allow inputs and outputs of data from and to external devices. Again, these instructions are hardware orientated, and will only be discussed briefly here.

IN A,(n)	**IN**put contents of port **n** to the **A**ccumulator

A 'port' is a connector to an outside device. Which number refers to which port is determined by hardware – i.e by the actual physical wiring and circuitry of the computer.

The port, as far as the Z80 is concerned, is an 8–bit data store. IN A,(n) does not affect any of the flags.

There is a form of the IN command which allows the contents of a port whose number is held in the C register to be placed in any of A, B, C, D, E, H or L:

IN r,(C)	**IN**put contents of port addressed by **C** to register **r**

This instruction resets the add/subtract flag. The carry flag is not affected, but the other flags are altered according to the value INput.

Should they be necessary, the Z80 also has forms of the IN command which parallel LDI, LDIR etc. The first of these is:

INI	**IN**put contents of port addressed by **C**, decrement B, **I**ncrement HL

With this instruction, B can be used as a loop counter if required. Notice, however, that only B is used by INI, rather than BC as in, say, LDI. C should contain the port number, and HL the memory location in which the data is to be placed. INI corrupts the sign, half–carry and parity/overflow flags, and sets the add/subtract flag. The carry flag is not affected. The zero flag is set when B=1.

INIR	**IN**put contents of port addressed by C, decrement B, **I**ncrement HL, **R**epeat until B=0

This is the same as INI except that it automatically repeats until B is decremented to zero. The flags are affected in the same way as with INI except that the zero flag is always set.

IND	**IN**put contents of port addressed by C, decrement B, **D**ecrement HL

INDR	**IN**put content of port addressed by C, decrement B, **D**ecrement HL, **R**epeat until B=0

These are the same as INI and INIR respectively, except that HL is decremented instead of incremented.

OUT (n),A	**OUT**put contents of **A**ccumulator to port **n**

OUT (C),r	**OUT**put contents of register **r** to the port addressed by **C**

The latter of these two differs from IN r,(C) in that it does not affect any of the flags.

OUTI	**OUT**put to port addressed by C the contents of memory addressed by HL, decrement B, Increment HL

OTIR	**OUT**put to port addressed by C the contents of memory addressed by HL, decrement B, Increment HL, Repeat until B=0

OUTD	**OUT**put to port addressed by C the contents of memory addressed by HL, decrement B, **D**ecrement HL

OTDR	**OUT**put to port addressed by C the contents of memory addressed by HL, decrement B, **D**ecrement HL, **R**epeat until B=0

These instructions affect the flags in the same way as the corresponding IN instructions.

One slightly useful instruction hasn't been mentioned yet:

NOP

NOP	No OPeration

This tells the Z80 to do nothing for one clock cycle. This can be useful for fine tuning the duration of delay loops. Also, if you are using absolute addresses rather than symbolic labels, it can be used to pad out your program to allow for addressing errors.

Summary

This chapter covered the following instructions:

DI	EX AF,AF'
EI	EX DE,HL
RETN	EX (SP),HL
RETI	EX (SP),IX
IM0	EX (SP),IY
IM1	HALT
IM2	IN A,(n)
RST n	IN r,(C)
EXX	INI
OUT (n),A	INIR
OUT (C),r	IND
OUTI	INDR
OTIR	NOP
OUTD	
OTDR	

Chapter 10

Einstein Graphics

This Chapter explains how to access the built–in graphics commands associated with the Einstein. Crystal Basic, which is supplied with the Einstein, has several 'built–in' functions, which allow the programmer to draw ELLIPSEs, POLYgons,LINEs and FILL areas on the screen. We shall investigate how such functions are carried out using assembly language.

Sprites are also an important feature of the Einstein's graphics capabilities, and a section on Sprite programming will also be included. Using Sprites involves looking up data from a table; this data is used to define the sprite. An essential part of such programmes is the use of tables and the HL register pair to step through these data tables and retrieve the data. An early example using text output will help to illustrate the method by which tables of data are handled.

Graphics Functions

The first graphics function which we will investigate is the ELLIPSE function, MCAL &C9. In the list of MCALS found in Appendix 4, the scratch pad locations used to hold various important pieces of required data, are as follows:

Scratch Pad Address		Function Performed
&FBA2	(HORRAD)	Holds Horizontal Radius
&FBA4	(VERRAD)	Holds Vertical Radius
&FBA6	(NUMSID)	Holds the number of sides: 100=Square; 04=Circle; 80 =Octagon
&FBA8	(DOTON)	Set to &FF for continuous line
&FBA9	(DOTOFF)	Set to 00 (refer to MCAL &C5)
&FBAA	(DOTON2)	Set to 00 (refer to MCAL &C5)
&FBAB	(DOTOF2)	Set to 00 (refer to MCAL &C5)
&FB9E	(STARTX)	Start point (x) of figure
&FBAØ	(STARTY)	Start point (y) of figure

The following program shows how these scratch pad locations and MCAL &C9 are used to draw a circle on the screen.

It uses a new assembler directive:

EQU nn	Assign the value **nn** to the preceding label.

In other words, a label, such as LABEL: is given a value; whenever the assembler sees the label thereafter, the corresponding value is used.

PROGRAM 10.1

```
HORRAD:  EQU &FBA2        Scratchpad location for horizontal radius.
VERRAD:  EQU &FBA4        Scratchpad location for vertical radius.
NUMSID:  EQU &FBA6        Number of sides to figure.
SQUARE:  EQU 1ØØ          Set NUMSID: to &100 for a square.
CIRCLE:  EQU 4            Set NUMSID: to 4 for a circle.
OCTAGN:  EQU 8Ø           Set NUMSID: to &80 for an octagon.
DOTON:   EQU &FBA8        Continuous line switch (scratch pad).
TYPE:    EQU &FF          Set DOTON to FF for a continuous line.
STARTX:  EQU &FB9E        Start point (x) of figure.
STARTY:  EQU &FBAØ
         LD HL,STARTX:
         LD A,1ØØ         Start point (y) of figure.
         LD (HL),A
         LD HL,STARTY:
         LD (HL),A
         LD HL,DOTON:
         LD A,TYPE:       Start point set to 100,100.
         LD (HL),A
         LD HL,&FBA9      Choose a continuous line.
         LD A,Ø           Remember to set the rest to 00.
         LD (HL),A        Refer to MCAL 5 if in doubt.
         LD HL,&FBAA
         LD (HL),A
         LD HL,&FBAB
         LD (HL),A
         LD HL,HORRAD:    Set up horizontal radius.
         LD A,5Ø
         LD (HL),A
         LD HL,VERRAD:    And vertical radius.
```

```
LD (HL),A
LD HL,NUMSID:
LD A,4
LD (HL),A
RST 8
DEFB &C9
RET
```

Choose the type of figure to be drawn.

In this case a circle.

At last the MCAL to draw a circle.

Wow! Time for a cup of tea.

Note the use of LABELS followed by the EQUates pseudo–operand to make the various scratchpad locations more meaningful. Defining memory locations in this way is much easier than having to figure out what value needs to go where. Since labels and EQUates statements do not form part of the Z80 instruction set, they cannot be converted into OP–CODES by any assembler. They are merely used by programmers as convenient tools for making the source code listings easier to follow; in the op–code, the labels will be replaced by the relevant values.

EXERCISE 10.1

Amend Program 10.1 to draw a rectangle on the screen. Remember that NUMSID: must now be set to &100

A possible solution is shown below.

PROGRAM 10.2

```
LD HL,SQUARE:
LD (NUMSID:),HL
LD (HL),A
RST 8
DEFB &C9
RET
```

These replace the last six lines of Program 10.1 above

The changes made to program 10.1 involve loading a two–byte value into the NUMSID: location. This cannot be done using the A register, since &100 is TWO bytes long. We therefore MUST use a 16 bit register, and the HL pair suits this purpose. Thus, by LoaDing HL with &100 (SQUARE:) and then *transferring* the *contents* of the HL pair to location NUMSID: we have achieved our objective. All that remains is to draw the square. Experiment with the various numbers associated with the graphics drawing MCALS and see how changing the parameters affects the figures produced.

Fill Routines

The ability to fill areas with foreground or background colour is a feature which is supported by the Einstein. As we might expect, the designers carefully built in to the operating system some useful routines to enable programmers to quickly and easily use this facility.

The MCAL &CD fills an area on the screen surrounding coordinates passed in the IX and IY register representing the X & Y coordinates to start at.

If the fill is to be in the same colour as the containing lines (in our case in WHITE), then the fill is to be in BACKGROUND colour, then FILMOD: MUST be set to zero.

Program 10.3 can be added to the bottom of Program 10.2, before the RET, to fill the diamond with foreground colour (WHITE).

PROGRAM 10.3

```
FILMOD: EQU &FBAD
LD IX,100          x coordinate of centre of diamond.
LD IY,100          y coordinate of centre of diamond.
LD A,&FF           set FILMOD to FOREGROUND.
LD (FILMOD:),A
RST 8
DEFB &CD           FILL MCAL.
RET
```

EXERCISE 10.2

Modify Program 10.3 to include an UNFILL section to erase the filled diamond on the screen.

A possible solution is given below.

PROGRAM 10.4

```
        RST 8
        DEFB &C9                    As Program 10.2.
        CALL FILL:                  Replace FILL MCAL by a CALL.
        CALL UNFILL:                CALL UNFILL:.
        RET                         Return to DOS.
FILL:   LD IX,100                   Now do the FILL.
        LD IY,100
        LD A,&FF
        LD (FILMOD:),A
        RST 8
        DEFB &CD                    Fill mcal.
        RET                         Return from fill.
UNFILL: LD IX,100
        LD IY,100
        LD A,0
        LD (FILMOD:),A
        RST 8
        DEFB &CD                    Fill MCAL.
        RET                         Return from unfill.
```

By now, we should be confidently handling MCALS, graphics routines, and handling FILL. The next logical step is to handle sprites, but just before we start on sprites proper, let's jut have a brief look at handling tables of data. We will use text in our data tables for the sake of simplicity.

Program 10.5 below uses the HL register pair to 'point' to a message held in memory. The exact location of this message is of no concern to us, since we label its start address with our assembler. This makes life very easy. Upon execution, Program 10.5 will print a message to the screen. As you can see, the program checks for the end of the message by looking for the ASCII code &80.

PROGRAM 10.5

```
CLS:     EQU &C              Clear screen.
         JR START:
CLEAR:   LD A,CLS:
         RST 8
         DEFB &9E            Execute CLS.
         RET                 end of CLEAR.
START:   CALL CLEAR:
         LD HL,MSG:          Load MSG address.
         LD A,&8Ø            Text terminator.
CHECK:   CP (HL)             Is character terminator?
         JR Z,DONE:          If yes finish.
         CALL OUTPUT:        Send to screen.
         INC HL              If no move counter.
         JR CHECK:           Get next char.
OUTPUT:  LD A,(HL)           Load a char.
         RST 8
         DEFB &9E            Call ZOUT MCAL.
         LD A,&8Ø            Re-load A.
         RET                 End of sub OUTPUT.
MSG:     DEFM "HELLO",&8Ø    Text & terminator.
DONE:    RET
```

After running the above program, the word HELLO will appear on the screen, having been called from some unknown location in memory whose address was determined by the HL register pair and the label MSG:

The MSG: label essentially points to a block of data; thus by loading the HL pair using the statement LD HL,MSG:, we are effectively labelling data blocks. This technique will be very useful when we come to handling sprites.

Sprite Handling in Assembly Language

The Texas Instruments (®) processor which is fitted to the Einstein (and to MSX Machines) allows up to 32 Sprites to be defined, and up to 4 Sprites to be present on ONE LINE at any time. Sprites can be moved around the screen quite simply by adjusting the screen coordinates of the upper left boundary of the sprite. Various priorities may be assigned to Sprite, and thus the appearance of 3–D movement can be achieved.

The first step is to define the sprite. The sprite might be a 'space invader' type character, or any other character you choose. The graphics processor requires the following information about any sprite:

(1)	the x coordinate	(1 byte)
(2)	the y coordinate	(1 byte)
(3)	the shape number	(1 byte)
(4)	colour information	(1 byte)

The four pieces of information concerning each sprite are known as the sprite's **ATTRIBUTES**, and are kept in **VIDEO–RAM** in a table which, not surprisingly, is known as the **Sprite Attribute Table.**

The sprite attribute table is fixed in memory and starts at location &3BØØ.

If we took a snapshot of the sprite attribute table, it would look something like this:

Sprite No	Byte 1	Byte 2	Byte 3	Byte 4
memory Loc. 0	&3BØØ **X–coord**	&3BØ1 **Y–coord**	&3BØ2 **Shape No**	&3BØ3 **Colour Data**
1	&3BØ4 **X–coord**	&3BØ5 **Y coord**	&3BØ6 **Chape No**	&3BØ7 **Colour Data**
2	Etc	Etc	Etc	Etc

This table reserves enough space in memory to define the attributes for all 32 sprites.

This tells us where to put the sprite, and what colour it is, but it does not define the shape of the sprite (unless we choose to make a sprite of one of the pre–defined keyboard characters – which we will illustrate shortly).

Sprite Shapes

The shape definition in the BASIC Reference Manual (page 209) illustrates quite well how the fundamental shapes for characters are made up from an 8*8 cell matrix. How these shapes are converted into numbers which the computer can use is further dealt with on pages 149–167 in the Introduction to Einstein manual. The details of the pattern of each sprite are again held in a table, called the**Pattern Generator Table**.

This table starts in V–RAM (Video RAM) at memory location &1800; for a simple sprite formed from an 8 x 8 cell matrix it requires eight bytes per sprite. Later we will see that this table is filled with sprite pattern information, but in the first example of sprite handling, we will choose, for simplicity's sake, to take the pre–defined character 65 (ASCII "A") and make it into a spite. We will then move it to screen location X=100, Y=100, at the same time making its colour black.

PROGRAM 10.6

```
ATTR:   EQU &3BØØ       Label the start of the attribute table.
        LD BC,ATTR:     Load BC with the start address of table.
        CALL SETADR:    This is how the data gets to the screen.
        LD A,1ØØ        X coordinate (stored at &3B00).
        OUT (8),A       Used by the MCAL to send to data via port 8
        EX (SP),HL
        EX (SP),HL      A slight delay to allow graphics chip time.
        LD A,1ØØ        Y coord (stored at &3B01).
        OUT (8),A       Send the Y coordinate.
        EX (SP),HL
        EX (SP),HL      Another delay for graphics chip.
        LD A,65         Load "A" into acc (this is the sprite no. byte).
        OUT (8),A       Send this byte to location &3B02.
        EX (SP),HL
        EX (SP),HL      Another 'breather' for graphics chip.
        LD A,1          The colour info. for location &3B03.
        OUT (8),A       Send colour byte to table.
        RET             Return to DOS.
SETADR: PUSH DE         Save the DE registers from corruption.
        RST 8           Call the MCAL to send data to the table.
        DEFB &C1        This is the actual MCAL used.
        POP DE          Restore the DE from stack.
        RET             Return from SETADR.
```

Now for a brief explanation of what's been going on. MCAL &C1 i used to send the address held in the BC pair to the Visual Display Processor (VDP). Data can then be output to V–RAM from PORT 8 (a channel by which the Einstein's CPU can communicate with other devices, in this case the VDP chip).

The first piece of the program sets up the sprite attribute table address, and then uses MCAL C1 to set up the Einstein's link to the VDP. All subsequent data bytes sent via port 8 to the VRAM will then be loaded into subsequent VRAM locations. This is very handy, since the next four bytes of data (X,Y,Shape,Colour), can then be sent directly to the Attribute table without having to bother about where in the table they will go. Having sent the X & Y coordinates to the Attribute table, the program sends a '65' or ASCII 'A'. In other words, we chose not to re–define our sprite, but instead to take the pre–defined letter 'A' and make it into sprite 0. This relieves us of the responsibility of loading the patterns into the Pattern Generator table, since it was already there.

The delays are necessary between all V–RAM reads and/or writes. If you choose not to have these delays, the sprite may be corrupted.

Now let's actually define a sprite and put it onto the screen.

First, we must decide on the shape of the sprite.

Figure 10.1 below shows an 8 x 8 cell matrix, alongside which is the hex code derived from the filled–in squares. The simplest way to arrive at these hex codes, is to imagine the 8 x 8 cell matrix broken down into four smaller units (as shown). Number each unit from right to left 1, 2, 4 and 8 respectively. Add up all the coloured–in units on each line and you will get the hex code for that component of the shape.

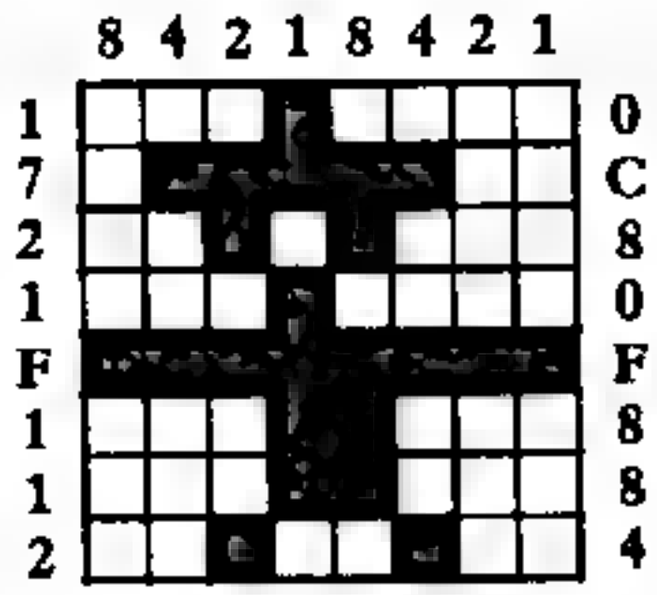


Figure 10.1

Having obtained our 'Gremlin' shape, let's put it into the Pattern Generator table, which starts at location &1800. Again, a look at the table for an 8 x 8 cell matrix, will produce the following.

Sprite No	Memory Location	Memory Location
Ø	&18ØØ	&1Ø
Ø	&18Ø1	&7C
Ø	&18Ø2	&28
Ø	&18Ø3	&1Ø
Ø	&18Ø4	&FF
Ø	&18Ø5	&18
Ø	&18Ø6	&18
Ø	$18Ø7	&24

1 If we wished to define another sprite, its data would be loaded into the *next* 8 bytes of the table, and so on.

Program 10.7 defines the above 'gremlin' and displays it at 100,100.

PROGRAM 10.7

```
PGEN:   EQU &18ØØ
ATTR:   EQU &3BØØ
        CALL SHAPE:          Load shape zero at 1800.
        LD BC,ATTR:          Load BC with table address.
        CALL SETADR:         Set up the pattern table address.
        LD A,1ØØ             X coordinate.
        OUT (8),A            Send to data port.
        EX (SP),HL  }
        EX (SP),HL  }        Slight delay.
        LD A,1ØØ
        OUT (8),A
        EX (SP),HL  }
        EX (SP),HL  }        Another delay.
```

```
        LD A,0                  Load acc with shape 0.
        OUT (8),A
        EX (SP),HL  }           Yet another delay.
        EX (SP),HL  }
        LD A,1
        OUT (8),A
        RET                     Return to DOS.
SETADR: PUSH DE
        RST 8                   MCAL to set up either pattern.
        DEFB &C1                Table or the attribute table.
        POP DE
        RET                     Return from setadr.
DELAY:  RET
SHAPE:  LD BC,PGEN:
        CALL SETADR:            Set up the start address of pattern table.
        LD HL,BITS:             The label for the sprite data counter (8 bytes to send).
        LD B,8
LOOP1:  LD A,(HL)               Loop to step through table.
        OUT (8),A
        INC HL
        DEC B
        EX (SP),HL  }           Delay.
        EX (SP),HL  }
        JP NZ,LOOP1:            Test if 8 bytes loaded.
        RET                     Return from SHAPE:.
BITS:   DEFB &10
        DEFB &7C,&28,&10,&FF,&18,&18,&24
```

Assemble and run the program above, and the little gremlin will appear on the screen at 100,100.

Try experimenting with the colour byte, to alter the colour of the sprite.

EXERCISE 10.3

Modify Program 10.7 to move the sprite diagonally across the screen from 0,0 until it runs off the screen.

A possible solution is given below.

PROGRAM 10.8

```
PGEN:   EQU &18ØØ
ATTR:   EQU &3BØØ
        CALL SHAPE:          Load shape zero at 1800
        LD H,&CF             Dist to move gremlin
        LD D,Ø               Start x
        LD E,D               Start y
LOOP:   LD BC,ATTR:
        CALL SETADR:
        LD C,8               Output port
        OUT (C),E            Write x coord
        EX (SP),HL
        EX (SP),HL
        OUT (C),D            Write y coord
        CALL DELAY:          Slow it down to see it
        LD A,Ø               Sprite number
        OUT (C),A            Send sprite 0 to table
        EX (SP),HL
        EX (SP),HL
        LD A,1               Load colour
        OUT (C),A            Send colour
        INC D                Move the x coord
        INC E                Move the y coord
        DEC H                Dist=dist-1
```

```
        JR NZ,LOOP:
        RET                          Back to DOS
SETADR: PUSH DE
        RST 8
        DEFB &C1
        POP DE
        RET                          Return from setadr
DELAY:  PUSH HL                      Save HL
        LD HL,15ØØ                   Delay factor
DEL1:   DEC HL
        LD A,H
        OR L                         Test HL for zero
        JR NZ,DEL1:
        POP HL                       Restore HL from zero
        RET                          Return from delay
SHAPE:  LD BC,PGEN:
        CALL SETADR:
        LD HL,BITS:
        LD B,8                       8 bytes to send
LOOP1:  LD A,(HL)
        OUT (8),A
        INC HL
        DEC B
        EX (SP),HL
        EX (SP),HL
        JP NZ,LOOP1:
        RET
BITS:   DEFB &1Ø
        DEFB &7C,&28,&1Ø,&FF,&18,&18,&24     Data for sprite 0
```

Well, that's it. You have made it to the end. Well done. Hopefully, you have enjoyed reading this book and feel that it was worthwhile. Assembly language programming may not be the simplest method of programming, but it certainly is one of the most sought–after skills within the computer world. Preserve, practice and modify your programs until they are code–efficient. Those who wish can take many examples from this book, particularly in Chapter 10, and reduce the amount of code considerably. Exercises of this kind are most beneficial, so please keep on with what must be one of the most exciting languages available. Happy programming!

Appendix 1

The Z80 Instruction Set

Abbreviations used in the following tables:

Registers

r,r'	= any one of A, B, C, D, E, H, L
dd	= (destination) any one of BC, DE, HL, SP
ss	= (source) any one of BC, DE, HL, SP
qq	= any one of the register pairs AF, BC, DE, HL
pp	= any one of the register pairs BC, DE, IX, SP
rr	= any one of the register pairs BC, DE, IY, SP
d	= a two's complement offset (–128 to 127)
e	= a two's complement offset (–126 to 129)
s	= any one of r, n, (HL), (IX+d), (IY+d)
m	= any one of r, (HL), (IX+d), (IY+d)
SP	= Stack Pointer
I	= Interrupt register
R	= Refresh register
b	=bit (0 to 7)

Addressing Modes

RR	= Register – Register
Im	= Immediate
IDX	= Indexed
D	= Direct
In	= Indirect

Flags

C	= Carry/link flag (also c and CY)
Z	= Zero flag
S	= Sign flag
P/V	= Parity or overflow flag
H	= Half–carry flag
N	= Add/Subtract flag

Key to flag outcomes

?	Flag set depending upon outcome of operation
0	Flag reset
1	Flag set
*	Flag not affected
–	Flag outcome unknown
V	Flag set on overflow
F	The P/V flag is set to the contenof the interrupt enable flip–flop (IFF)

The P/V Flag

If the operation indicated by a V in the P/V column results in an overflow then the V flag will be set (1) otherwise it will be reset (0). If the operation is used to test the parity, indicated by a P, then the flag will be set (1) if the parity is even else reset (0) on parity odd. A ? in this column means that the instruction uses this flag in a special way (see relevant chapter).

Addressing Modes

LD r,r'	Register–register
LD r,n	Immediate
LD r,(IX+d)	Indexed
LD r,(nn)	Direct
LD r,(ss)	Indirect

Where

r or r' is a 8–bit register
n is an 8–bit number
d is a 2's complement offset
nn is a 16–bit number
ss is one of BC, DE, HL, SP

Eight–Bit Load Group

Mnemonic	Symbolic operation	No of Bytes	No of cycles	Addr mode	Flags Affected C Z P/V S N H
LD r,r'	r ← r'	1	1	RR	* * * * * *
LD r,n	r ← n	2	2	Im	* * * * * *
LD r,(HL)	r ← (HL)	1	2	In	* * * * * *
LD r,(IX+d)	r ← (IX+d)	3	5	IDX	* * * * * *
LD r,(IY+d)	r ← (IY+d)	3	5	IDX	* * * * * *
LD (HL),r	(HL) ← r	1	2	In	* * * * * *
LD (IX+d),r	(IX+d) ← r	3	5	IDX	* * * * * *
LD (IY+d),r	(IY+d) ← r	3	5	IDX	* * * * * *
LD (HL),n	(HL) ← n	2	3	In	* * * * * *
LD (IX+d),n	(IX+d) ← n	4	5	IDX	* * * * * *
LD (IY+d),n	(IY+d) ← n	4	5	IDX	* * * * * *
LD A,(BC)	A ← (BC)	1	2	In	* * * * * *
LD A,(DE)	A ← (DE)	1	2	In	* * * * * *
LD A,(nn)	A ← (nn)	3	4	D	* * * * * *
LD (BC),A	(BC) ← A	1	2	In	* * * * * *
LD (DE),A	(DE) ← A	1	2	In	* * * * * *
LD (nn),A	(nn) ← A	3	4	D	* * * * * *
LD A,I	A ← I	2	2	RR	* ? F ? 0 0
LD A,R	A ← R	2	2	RR	* ? F ? 0 0
LD I,A	I ← A	2	2	RR	* * * * * *
LD R,A	R ← A	2	2	RR	* * * * * *

Sixteen–Bit Load Group											
Mnemonic	**Symbolic operation**	**No of Bytes**	**No of cycles**	**Addr mode**	**Flags Affected** C	Z	P/V	S	N	H
LD dd,nn	dd ← nn	3	3	Im	*	*	*	*	*	*
LD IX,nn	IX ← nn	4	4	Im	*	*	*	*	*	*
LD IY,nn	IY ← nn	4	4	Im	*	*	*	*	*	*
LD HL,(nn)	H ← (nn+1) L ← (nn)	3	5	D	*	*	*	*	*	*
LD dd,(nn)	dd ← (nn+1) dd ← (nn)	4	6	D	*	*	*	*	*	*
LD IX,(nn)	IX ← (nn+1) IX ← (nn)	4	6	D	*	*	*	*	*	*
LD IY,(nn)	IY ← (nn+1) IY ← (nn)	4	6	D	*	*	*	*	*	*
LD (nn),HL	(nn+1) ← H (nn) ← L	3	5	D	*	*	*	*	*	*
LD (nn),dd	(nn+1) ← dd (nn) ← dd	4	6	D	*	*	*	*	*	*
LD (nn),IX	(nn+1) ← IX (nn) ← IX	4	6	D	*	*	*	*	*	*
LD (nn),IY	(nn+1) ← IY (nn) ← IY	4	6	D	*	*	*	*	*	*
LD SP,HL	SP ← HL	1	1	RR	*	*	*	*	*	*
LD SP,IX	SP ← IX	2	2	RR	*	*	*	*	*	*
LD SP,IY	SP ← IY	2	2	RR	*	*	*	*	*	*
PUSH qq	(SP–1) ← qq (SP–2) ← qq	1	3	Im	*	*	*	*	*	*
PUSH IX	(SP–1) ← IX (SP–2) ← IX	2	4	Im	*	*	*	*	*	*
PUSH IY	(SP–1) ← IY (SP–2) ← IY	2	4	Im	*	*	*	*	*	*
POP qq	qq ← (SP) qq ← (SP+1)	1	3	Im	*	*	*	*	*	*
POP IX	IX ← (SP) IX ← (SP+1)	2	4	Im	*	*	*	*	*	*
POP IY	IY ← (SP) IY ← (SP+1)	2	4	Im	*	*	*	*	*	*

Exchange, Block Transfer and Serach groups

Mnemonic	Symbolic operation	No of Bytes	No of cycles	Addr mode	Flags Affected C	Z	P/V	S	N	H
EX DE,HL	DE ↔ HL	1	1	RR	*	*	*	*	*	*
EX AF,AF'	AF ↔ AF'	1	1	RR	*	*	*	*	*	*
EXX	[BC] ↔ [BC'] [DE] ↔ [DE'] [HL] ↔ [HL']	1	1	RR	*	*	*	*	*	*
EX (SP),HL	H ↔ (SP+1) L ↔ (SP)	1	5	RR	*	*	*	*	*	*
EX (SP),IX	IX ↔ (SP+1) IX ↔ (SP)	2	6	RR	*	*	*	*	*	*
EX (SP),IY	IY ↔ (SP+1) IY ↔ (SP)	2	6	RR	*	*	*	*	*	*
LDI	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC–1	2	4	In	*	*	?	*	0	0
LDIR	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC–1 until BC=0	2	5	In	*	*	0	*	0	0
LDD	(DE) ← (HL) DE ← DE–1 HL ← HL–1 BC ← –1	2	4	In	*	*	?	*	0	0
LDDR	(DE) ← (HL) DE ← DE–1 HL ← HL–1 BC ← BC–1 until BC=0		5	In	*	*	0	*	0	0

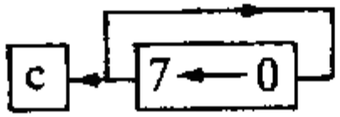
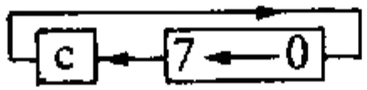
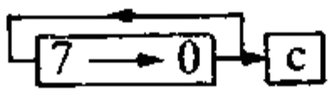
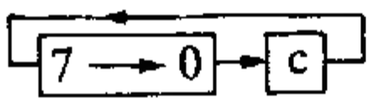
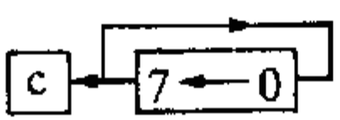
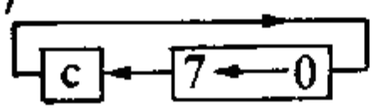
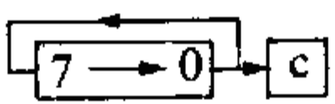
Mnemonic	Symbolic operation	No of Bytes	No of cycles	Addr mode	Flags Affected C	Z	P/V	S	N	H
CPI	A–(HL) HL ← HL+1 BC ← BC–1	2	4	In	*	?	?	?	?	?
CPIR	A–(HL) HL ← HL+1 BC ← BC–1 until A=(HL) or BC=0	2	5	In	*	?	?	?	?	?
CPD	A–(HL) HL ← HL–1 BC ← BC–1	2	4	In	*	?	?	?	?	?
CPDR	A–(HL) HL ← HL–1 BC ← BC–1 until A=(HL) or BC=0	2	5	In	*	?	?	?	?	?

Eight–Bit Arithmetic and Logical Group												
Mnemonic	**Symbolic operation**	**No of Bytes**	**No of cycles**	**Addr mode**	**Flags affected** **C**	**Z**	**P/V**	**S**	**N**	**H**
ADD A,R	A←A+r	1	1	Im	?	?	V	?	0	?
ADD A,n	A←A+n	2	2	Im	?	?	V	?	0	?
ADD A,(HL)	A←A+(HL)	1	2	In	?	?	V	?	0	?
ADD A,(IX+d)	A←A+(IX+d)	3	5	IDX	?	?	V	?	0	?
ADD A,(IY+d)	A←A+(IY+d)	3	5	IDX	?	?	V	?	0	?
ADC A,s	A←A+s+CY	1–3	1–5	Im	?	?	V	?	0	?
SUB s	A←A–s	1–3	1–5	Im	?	?	V	?	1	?
SBC A,s	A←A–s–CY	1–3	1–5	Im	?	?	V	?	1	?
AND s	A←A∩s	1–3	1–5	Im	0	?	P	?	0	1
OR s	A←A∪s	1–3	1–5	Im	0	?	P	?	0	1
XOR s	A←A⊕s	1–3	1–5	Im	0	?	P	?	0	1
CP s	A–s	1–3	1–5	Im	?	?	V	?	1	?
INC r	r← r+1	1	1	Im	*	?	V	?	0	?
INC (HL)	(HL)←(HL)+1	1	3	In	*	?	V	?	0	?
INC (IX+d)	(IX+d)← (IX+d)+1	3	6	IDX	*	?	V	?	0	?
INC (IY+d)	(IY+d)←(IY+d)+1	3	6	IDX	*	?	V	?	0	?
DEC m	m←m–1	1–3	1–6	Im	*	?	V	?	0	?

General Purpose Arithmetic and C.P.U. Control Groups

Mnemonic	Symbolic operation	No of Bytes	No of cycles	Addr mode	Flags affected C	Z	P/V	S	N	H
DAA	Converts A into packed BCD after an add or subtract on BCD operands	1	1	Im	?	?	P	?	*	?
CPL	$A \leftarrow \overline{A}$	1	1	Im	*	*	*	*	1	1
NEG	$A \leftarrow 0-A$	2	2	Im	?	?	V	?	1	?
CCF	$CY \leftarrow \overline{CY}$	1	1	Im	?	*	*	*	0	*
SCF	$CY \leftarrow 1$	1	1	Im	1	*	*	*	0	0
NOP	No operation	1	1	Im	*	*	*	*	*	*
HALT	CPU halted	1	1	Im	*	*	*	*	*	*
DI	$IFF \leftarrow 0$	1	1	Im	*	*	*	*	*	*
EI	$IFF \leftarrow 1$	1	1	Im	*	*	*	*	*	*
IM0	Set Interrupt mode 0	2	2	Im	*	*	*	*	*	*
IM1	Set Interrupt mode 1	2	2	Im	*	*	*	*	*	*
IM2	Set Interrupt mode 2	2	2	Im	*	*	*	*	*	*

Sixteen–Bit Arithmetic Group											
Mnemonic	**Symbolic operation**	**No of Bytes**	**No of cycles**	**Addr mode**	**Flags affected** C	Z	P/V	S	N	H
ADD HL,dd	HL ← HL+dd	1	3	Im	?	*	*	*	0	–
ACD HL,dd	HL ← HL+dd+CY	2	4	Im	?	?	V	?	0	–
SBC HL,dd	HL ← HL–dd–CY	2	4	Im	?	?	V	?	1	–
ADD IX,pp	IX ← IX+pp	2	4	Im	?	*	*	*	0	–
ADD IY,rr	IY ← IY+rr	2	4	Im	?	*	*	*	0	–
INC dd	dd ← dd+1	1	1	Im	*	*	*	*	*	*
INC IX	IX ← IX+1	2	2	Im	*	*	*	*	*	*
INC IY	IY ← IY+1	2	2	Im	*	*	*	*	*	*
DEC dd	dd ← dd–1	1	1	Im	*	*	*	*	*	*
DEC IX	IX ← IX–1	2	2	Im	*	*	*	*	*	*
DEC IY	IY ← IY–1	2	2	Im	*	*	*	*	*	*

Rotate and Shift group											
Mnemonic	**Symbolic operation**	**No of Bytes**	**No of cycles**	**Addr mode**	**Flags affected** C	Z	P/V	S	N	H
RLCA		1	1	Im	?	*	*	*	0	0
RLA		1	1	Im	?	*	*	*	0	0
RRCA		1	1	Im	?	*	*	*	0	0
RRA		1	1	Im	?	*	*	*	0	0
RLC r		2	2	Im	?	?	P	?	0	0
RLC (HL)		2	4	Im	?	?	P	?	0	0
RLC (IX+d)		4	6	Im	?	?	P	?	0	0
RLC (IY+d)		4	6	Im	?	?	P	?	0	0
RL m			–	Im	?	?	P	?	0	0
RRC m			–	Im	?	?	P	?	0	0

RR m		–	–	Im	?	?	P	?	0	0
SLA m		–	–	Im	?	?	P	?	0	0
SRA m		–	–	Im	?	?	P	?	0	0
SRL m		–	–	Im	?	?	P	?	0	0
RLD		2	5	Im	*	?	P	?	0	0
RRD		2	5	Im	*	?	P	?	0	0

Bit set, Reset and Test Group											
Mnemonic	**Symbolic operation**	**No of Bytes**	**No of cycles**	**Addr mode**	**Flags affected** **C**	**Z**	**P/V**	**S**	**N**	**H**
Bit 6,r	Z←r	2	2	RR	*	?	–	–	0	1
Bit 6,(HL)	Z←(HL)	2	3	In	*	?	–	–	0	1
Bit 6,(IX+d)	Z←(IX+d)	4	5	IDX	*	?	–	–	0	1
Bit 6,(IY+d)	Z←(IY+d)	4	5	IDX	*	?	–	–	0	1
Set 6,r	r←1	2	2	RR	*	*	*	*	*	*
Set 6,(HL)	(HL)←1	2	4	In	*	*	*	*	*	*
Set 6,(IX+d)	(IX+d)←1	4	6	IDX	*	*	*	*	*	*
Set 6,(IY+d)	(IY+d)←1	4	6	IDX	*	*	*	*	*	*
Res 6,s	s←0	–	–	RR	*	*	*	*	*	*

Jump Group											
Mnemonic	**Symbolic operation**	**No of Bytes**	**No of cycles**	**Addr mode**	**Flags affected**						
					C	**Z**	**P/V**	**S**	**N**	**H**
JP nn	PC ← n	3	3	Im	*	*	*	*	*	*
JP cc,nn	If condition cc is true PC nn otherwise continue.	3	3	Im	*	*	*	*	*	*
JR e	PC ← PC+e	2	3	Im	*	*	*	*	*	*
JR C,e	If c=0 continue.	2	2	Im	*	*	*	*	*	*
JR NC,e	If c=1 continue.	2	2	Im	*	*	*	*	*	*
JR Z,e	If z=0 continue.	2	2	Im	*	*	*	*	*	*
JR NZ,e	If z=1 continue.	2	2	Im	*	*	*	*	*	*
JP (HL)	PC ← HL	1	1	In	*	*	*	*	*	*
JP (IX)	PC ← IX	2	2	IDX	*	*	*	*	*	*
JP (IY)	PC ← IY	2	2	IDX	*	*	*	*	*	*
DJNZ e	B ← B–1 If B=0 continue. If B=0 PC ← PC+e	2	3	Im	*	*	*	*	*	*

Call and Return Group					
Mnemonic	**Symbolic operation**	**No of Bytes**	**No of cycles**	**Addr mode**	**Flags affected C Z P/V S N H**
CALL nn	(SP–1) ← PC (SP–2) ← PC PC ← nn	3	5	Im	* * * * * *
CALL cc,nn	If condition cc is false continue, otherwise same as CALL nn	3	3	Im	* * * * * *
RET	PC ← (SP) PC ← (SP+1)	1	3	Im	* * * * * *
RET cc	If condition cc is false continue, otherwise same as RET	1	1	Im	* * * * * *
RETI	Return from interrupt	2	4	Im	* * * * * *
RETN	Return from non markable interrupt	2	4	Im	* * * * * *
RST p	(SP–1) ← PC (SP–2) ← PC PC ← 0 PC ← P	1	3	Im	* * * * * *

Input and Output Group												
Mnemonic	**Symbolic operation**	**No of Bytes**	**No of cycles**	**Addr mode**	**Flags affected**							
					C	**Z**	**P/V**	**S**	**N**	**H**		
IN A,(n)	A←(n)	2	3	Im	*	*	*	*	*	*		
IN r,(c)	r←(c)	2	3	In	*	?	P	?	0	?		
INI	(HL)←(c) B←B–1 HL←HL+1	2	4	In	*	?	–	–	1	–		
INIR	(HL)←(c) B←B–1 HL← HL+1 until B=0	2	5	In	*	1	–	–	1	–		
IND	(HL)←(c) B←B–1 HL←HL–1	2	4	In	*	?	–	–	1	–		
INDR	(HL)←(c) B←B–1 HL←HL–1 until B=0	2	5	In	*	1	–	–	1	–		

Input and Output Group (contd.)												
Mnemonic	**Symbolic operation**	**No of Bytes**	**No of cycles**	**Addr mode**	**Flags affected** **C**	**Z**	**P/V**	**S**	**N**	**H**
OUT (n),a	(n)←a	2	3	Im	*	*	*	*	*	*
OUT (c),r	(c)←r	2	3	In	*	*	*	*	*	*
OUTI	(c)←(HL) B←B–1 HL← HL+1	2	4	In	*	?	–	–	1	–
OTIR	(c)←(HL) B←B–1 HL←HL+1 until B=0	2	5	In	*	1	–	–	1	–
OUTD	(c)←(HL) B←B–1 HL←HL–1	2	4	In	*	?	–	–	1	–
OTDR	(c)←(HL) B←B–1 HL←HL–1 until B=0	2	5	In	*	1	–	–	1	–

Appendix 2

Effects Of Instructions On The Flags

The Flags

Bit	Flag	
0	C	Carry
1	N	add/subtract
2	P/V	Parity/oVerflow
3	–	–
4	H	Half carry
5	–	–
6	Z	Zero
7	S	Sign

Effects Of Instructions on the Sign Flag

Group	Instruction;	Effects
8–bit Loads	LD A,I LD A,R	Set if I register is negative, else reset. Set if R register is negative, else reset.
Compares	CPI, CPIR, CPD, CPDR	Set if result is negative, else reset.
8–bit arithmetic	ADD A,S ADC A,S SUB S SBC A,S AND S OR S XOR S CP S INC S DEC S	Set if result is negative, else reset.
General purpose arithmetic	DAA NEG	Set if msb of A 1, else reset. Set if result is negative, else reset.
16–bit arithmetic	ADC HL,SS SBC HL,SS	Set if result is negative, else reset.

Shifts and rotates	RLC S RL S RRC S RR S SLA S SRA S SRL S	Set if result is negative, else reset.
	RLD RRD	Set if A is negative after shift, else reset.
Bit	BIT B,S	Corrupted
Inputs and outputs	IN R,(C)	Set if input data is negative, else reset.
	INR IND,INDR, OUTI,OTIR, OUTD,OTDR	Corrupted

Effects of Instructions on the Zero Flag

Group	Instruction;	Effects
8–bit loads	LD A,I LDA,R	Set Z if I register=0, else reset Z. Set Z if R register=0, else reset Z.
Compares	CPI,CPIR, CPD,CPDR	Set Z if A=(HL), else reset Z.
8–bit arithmetic	ADD A,S ADC A,S SUB S SBC A,S AND S OR S XOR S CP S INC S DEC S	Set if result is zero, else reset.
General purpose arithmetic	DAA NEG	Set Z if result 0, else reset Z.
16–bit arithmetic	ADC HL,SS SBC HL,SS	Set Z if result−0, else reset Z.

RLC S RL S Shifts and rotates	RRC S RR S SLA S SRA S SRL S RLD RRD	Set if result is zero, else reset.
Bit	BIT B,S	Set Z if specified bit—0, else reset Z.
Inputs and Outputs	INR,(C) INI,IND, INIR,INDR OUTI,OUTD OTIR,OTDR	Set Z if input data=0 else reset Z. Set Z if B–1=0, else reset Z. Set Z. Set Z if B–1=0, else reset Z. Set Z.

Effects Of Instructions on the Half–Carry and Add/Subtract Flags

Group	Instruction	Half–Carry	Add/Subtract
8–bit	LD A,I I.D A,R	Reset	Reset
Block loads	LDI,LDIR, LDD,LDDR	Reset	Reset
Compares	CPI,CPIR, CPD,CPDR	Set if no borrow from bit 4 else reset	Set
8–bit arithmetic	ADD A,S ADC A,S	Set if no carry from bit 3, else reset	Reset
	SUB S SBC A,S	Set if no borrow from bit 4, else reset	Set
	AND S OR S XOR S	Set	Reset
	CP S	Set if no borrow from bit 4, else reset	Set
	INC S	Set if carry from bit 3, else reset	Reset
	DEC S	Set if borrow from bit 4, else reset	Set

Group	Instruction	Half -Carry	Add/Subtract
General purpose arithmetic	DAA	Corrupted	No effect
	CPL	Set	Set
	NEG	Set if no borrow from bit 4, else reset	Set
	CCF	No effect	Reset
	SCF	Reset	Set
16–bit arithmetic	ADD HL,SS ADC HL,SS	Set if carry from bit 11, else reset	Reset
	SBC HL,SS	Set if no borrow from bit 12, else reset	Set
	ADD IX,PP ADD IY,RR	set if carry from bit 11, else reset	Reset
Shifts and rotates	RLCA RLA RRCA RRA RLC S RL S RRC S RR S SLA S SRA S SRL S RLD RRD	Reset	Reset
Bit	BIT B,S	Set	Reset
Inputs outputs	INR,(C)	Reset	Reset
	INI,INIR IND,INDR, OUTI,OTIR, OUTD,OTDR	Corrupted	Set

Effects of Instructions on the Parity/Overflow Flag

Group	Instruction	Effects
8–bit loads	LD A,I	Copy of interrupt flip–flop 2.
	LD A,R	
Block instructions	LDI,LDD CPI,CPIR CPD,CPDR	Set if BC<>1, else reset.
	LDIR,LDDR	Reset
8–bit arithmetic	ADD A,S ADC A,S SUB S SBC A,S	Set if overflow, else reset.
	AND S OR S XOR S	Set if parity even, else reset.
	CP S	Set if overflow, else reset.
	INC S	Set if operand was $7F_{16}$ before increment, else reset.
	DEC S	Set if operand was 80_{16} before increment, else reset.
General purpose arithmetic	DAA	Set if (A) parity even, else reset.
	NEG	Set if (A) was 80_{16} before negate, else reset.
16–bit arithmetic	ADC HL,SS SBC HL,SS	Set if overflow, else reset.

Group	Instruction	Effects
Shifts and rotates	RLC S RL S RRC S SLA S SRA S SRL S RLD S RRD S	Set if parity even, else reset.
Bit	BIT B,S	Corrupt
Inputs and Outputs	IN R,(C) INI,INIR, IND,INDR, OUTI,OTIR, OUTD,OTDR	Set if parity even, else reset. Corrupt

Effects of Instructions on the Carry Flag

Group	Instruction	Effects
8–bit loads	LD A,I LD A,R	No effect
Block intructions	CPI,CPIR, CPD,CPDR LDI,LDIR LDD,LDDR	No effect
8–bit arithmetic	ADD A,S	Set if carry from bit 7, else reset.
	ADC A,S	
	SUB S	Set if no borrow, else reset.
	SBC S	
	AND S	
	OR S XOR S	Reset
	CP S	Set if no borrow, else reset.
	INC S DEC S	No effect
General purpose arithmetic	DAA	Set if BCD carry, else reset.
	NEG	Set if A was not 0 before negate, else reset.
	CCF	Set if flag was 0 before CCF, else reset.
	SCF	Set

Group	Instruction	Effects
16–bit arithmetic	ADD HL,SS	Set if carry from bit 15, else reset.
	ADD LH,SS	
	SBC HL,SS	Set if no borrow, else reset.
	ADD IX,PP	Set if carry from bit 15, else reset.
	ADD IY,RR	
Shifts and rotates	RLCA RLA	Copy A bit 7.
	RRCA RRA	Copy A bit 0.
	RLC S RL S	Copy bit 7 of operand.
	RRC S RR S	Copy bit 0 of operand.
	SLA S	Copy bit 7 of operand.
	SRA S	Copy bit 0 of operand.
	SRL S	Copy bit 0 of operand.

Appendix 3

The Effects of Comparisons on the Overflow, Sign and Carry Flags

e.g. value		Interpretation		Flag		
A	s	8-bit	2's comp	V	S	C
3Ø	2Ø	A>s	A>s	Ø	Ø	Ø
2Ø	CØ	A<s	A>s	Ø	Ø	1
7Ø	CØ	A<s	A>s	1	1	1
CØ	7Ø	A>s	A<s	1	Ø	Ø
CØ	2Ø	A>s	A<s	Ø	1	Ø
2Ø	3Ø	A<s	A<s	Ø	1	1

Key

s = One of the following: A, B, C, D, E, H, L, (HL), (IX+d), (IY+d)

1 = flag set 0 = flag reset

Conditions:

Flag	Set	Reset
V	PO (Parity Odd)	PE (Parity Even)
S	P (Positive)	M (Negative)
C	C (Carry)	NC (No Carry)

In the above, if the sign and overflow flags are the same, then A>s in two's complement. If they are different, then A<s, assuming the zero flag is not set.

Appendix 4

Some Built–in Routines

Like many other computers, the Einstein has a ROM (Read Only Memory) containing routines which perform certain functions. One of these functions is to boot the disc when the Einstein is powered up; this is how the Einstein knows that we wish to use the machine. Generally, the ROM is split up into little sections of program, each of which does a certain task. Whilst these sections may be regarded as essentially independent programs, when put together and used to complement one another they give the Einstein its "personality".

The way in which Einstein uses the routines has already been described in the main body of this book in general terms, and a list of the main MCAL's now follows.

In the text which follows, the following abbreviations apply:

CTC	Counter Timer Circuit
FDC	Floppy Disc Controller
MCAL	Machine Call
PSG	Programmable Sound Generator
VDP	Visual Display Processor
VRAM	Video RAM.

The MCAL's are executed following a ReSTart 8 instruction, which is followed by the function number in HEXadecimal (e.g. as a DEFined Byte). After execution of the ReSTart, the program resumes execution at the point immediately after the MCAL was called.

The list of MCALs which follows refers to MOS 1.2 and assumes that the Einstein is being used in 40 Column Mode. All MCALs which output text data to the screen will continue to work if an 80–column card is present, *but* MCALs C3 to CD will still direct their output to the VRAM.

Einstein MOS 1.2 MCAL Information

Function Number	Source Label	Description
80	ARITH	Performs as 'A' (ARITHMETIC) from MOS. *Values Passed* xxxx in HL pair. yyyy in DE pair (see manual).
81	BAUD	Performs as 'B' (BAUD from MOS). *Values Passed* x – Receive rate – upper nybble of L register. y – Transmit rate – lower nybble of L register. ww – Mode Byte – D register. zz – Command Byte – E register (see manual). If DE is zero, the mode and command bytes will remain unchanged. For the correct recieve and transmit rates, the baud rate factor x 16 must be used.
82	COPY	Performs as 'C' (COPY) from MOS. *Values Passed* xxxx – Start – in HL pair. yyyy – Finish – in DE pair. zzzz – destination – in BC pair (see manual).

Function Number	Source Label	Description
83	DECIMAL	Performs as 'D' (DECIMAL) from MOS. *Values Passed* xxxx – in HL pair (see manual).
84	EXEC	Performs as 'E' (EXECUTE) from MOS. *Values Passed* xxxx – Break point – in HL pair (see manual).
85	MFILL	Performs as 'F' (FILL) from MOS. *Values Passed* xxxx – Start – HL pair yyyy – Finish – DE pair zz – Value – C register (see manual).
86	GOTO	Performs as 'G' (GOTO) from MOS. *Values Passed* xxxx – execution address – HL pair. yyyy – break point – DE pair (see manual). If DE is zero, no break point is set.

Function Number	Source Label	Description
87	HEX	Peforms as 'H' (HEXADECIMAL) from MOS. *Values Passed* Pointer to text (in RAM) – DE pair. The decimal number is held at (DE) and terminated with 00. The hexadecimal number is returned in the HL pair in addition to being displayed.
8C	MODIFY	Performs as 'M' (MODIFY) from MOS. *Values Passed* xxxx – address to modify from – HL pair.
91	RDBLOK	Perform as 'R' (READ) from MOS. *Values Passed* Pointer to text (in RAM) – DE pair. The text takes the format as shown in the manual (see ZRBLK function no. A4).

Function Number	Source Label	Description
93	TBLAT	Performs as 'T' (TABULATE) from MOS. *Values Passed* xxxx – Start address – HL pair. yyyy – finish address – DE pair. zz – no. of columns – C register. If C is passed as zero, zz will default to 8 columns.
96	WRBLOK	Perfoms as 'W' (WRITE) from MOS. *Values Passed* Pointer to text (in RAM) – DE pair. The text at (DE) takes the format as shown in the manual (see ZWBLK function no. A5).
97	COLD	Performs as 'X' (COLD START) from MOS.
98	WARM	Perfoms as 'Y' (WARM START) from MOS.

Function Number	Source Label	Description
99	REGSTR	Performs as 'Z' (REGISTER EXAMINE) from MOS. *Values Passed* x – registers to be displayed – L register (0, 1 or 2). Note: This does not display the current register contents but the register contents at the last RST 38 (&FF encountered.)
9A	ZINIT	Re–entry point to MOS.
9B	ZRSCAN	Repeat key scan. This will return, in the A register, the value of any key pressed. 00 is returned if no key is pressed. 00 can also be returned if the keyboard poll rate (polled with this MCAL) is greater than the key repeat speed. This repeat speed can be altered in scratch pad location &FB43. **Note**: This works as the KBD command in XBAS.

Function Number	Source Label	Description
9C	ZKEYIN	Input key. This will return, in the A register, the value of any key pressed. Unlike MCAL 9B, this will wait for a key to be pressed. **Note:** This is similar to the INCH command in XBAS.
9D	ZGETLN	Get Text from keyboard. This will enter a line of text from the keyboard into RAM from the address held in the DE pair. The text is displayed on the screen on pressing the keys and the line is terminated with an ENTER.
9E	ZOUTC	Character Output. Outputs a character held in the A register to the screen.
9F	ZPOUT	Outputs a character to the parallel printer held in the A register.

Function Number	Source Label	Description
A0	ZSLOUT	Serial Output. Outputs a character to the serial port from the A register.
A1	ZSRLIN	Serial Input. Read a byte from the serial port and returns the value in the A register.
A2	ZRSECT	Sector Read. Reads a sector from the disc into the sector buffer (a sector is 200_{16} bytes). The following are set up in the scratchpad. **Location Label** &FB50 HSTDSC – Disc drive (0 – 3). &FB51 HSTTRK – Track (0 – &27). &FB52 HSTSEC – Sector (0 – 9). &FB53 HSTDMA – Sector buffer address (normally &FE00).
A3	ZWSECT	Sector Write. Writes a sector from the disc into the sector buffer (see MCAL A2).

Function Number	Source Label	Description
A4	ZRBLK	Read Block Read a block of data from the disc. *Values Passed* Drive no. (0–3) in A register. Start address in HL pair. Finish address in DE pair. Sector (0–9) in B register. Track (0–&27) in C register. Memory is filled to the next complete sector (200H bytes) e.g. if the start and finish address is specified as &6000 and &6001 respectively, &6000 to &6200 will be read from the disc.
A5	ZWBLK	Write Block Writes a block of data to the disc. The values passed are the same as for MCAL A4 and again is written to the next complete sector (&200 bytes).
A6	ZCRLF	Outputs a CR (Carriage Return &D) and LF (Line Feed &A).
A7	ZCRLFZ	Outputs a CR and LF if the cursor is not at column zero.

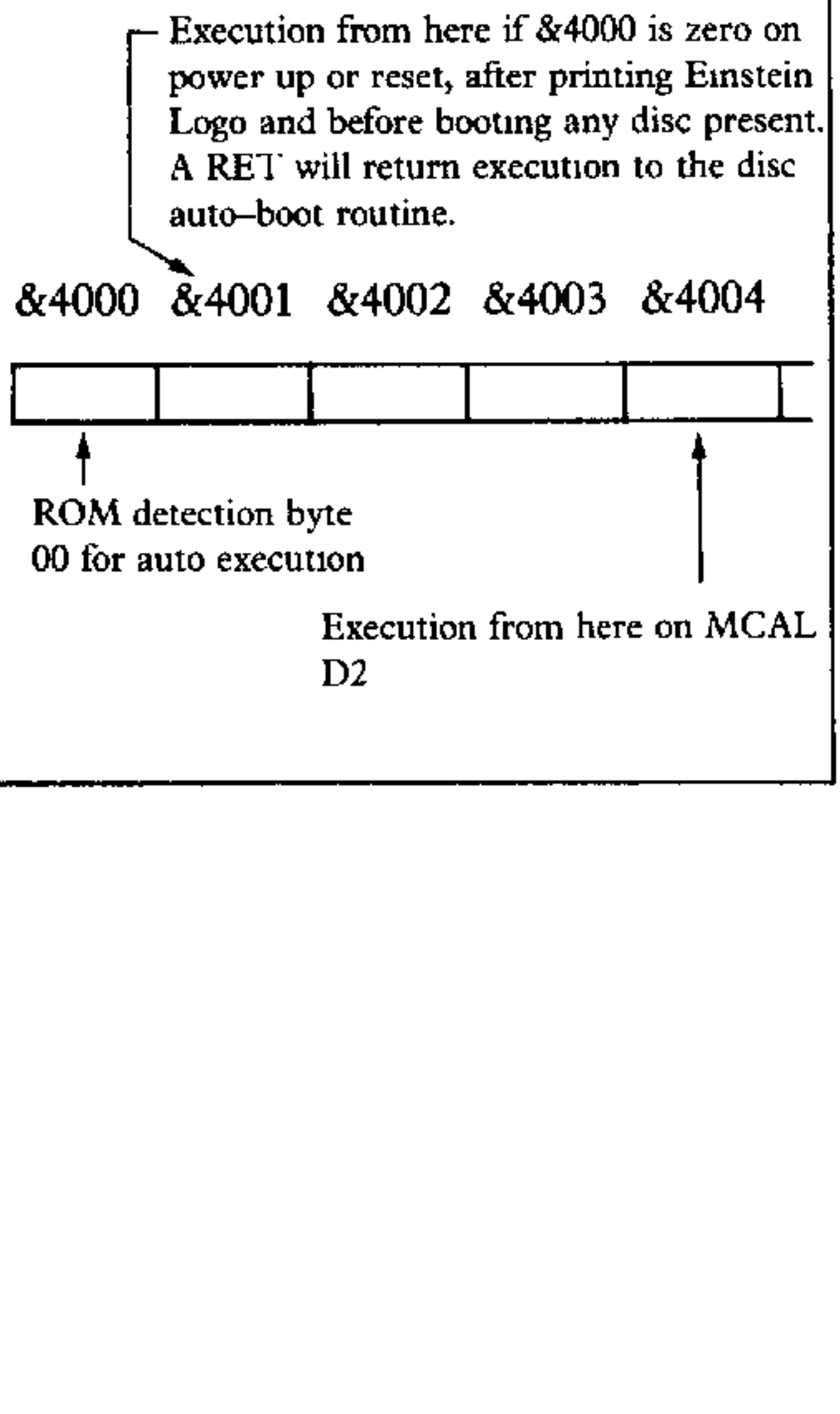
Function Number	Source Label	Description
A8	ZSPACE	Outputs 1 space.
A9	ZPR4HX	Outputs 4 hex digits held in the HL pair. e.g. if HL = &1234, 1234 is output.
AA	ZPR2HX	Outputs two hex digits held in the A register followed by a space.
AB	ZPH2HX	Outputs two hex digits held in the A register (as MCAL AA with no space output).
AC	ZFC4HX	Get a hex number (upto 4 digits) from text into the HL pair. DE points to the text (in RAM). The number is terminated on a non hex character.
AD	ZFCZHX	Get a hex number (upto 2 digits) from text into the A register. DE points to the text. The number is terminated with a non–hex character.

Function Number	Source Label	Description
AE	ZDCMD	Outputs a command to the FDC. The command type is passed in the A register. On return A is set to 00 unless the FDC is not executing the command then A is set to &FF.
AF	ZHMDSC	Takes the drive head to track 00. The drive no. is passed in the A register.
B0	ZIGBLK	Returns a value in the A register from RAM pointed to by the DE pair if DE>&7FFF, or from ROM if DE<&8000. **Note**: Commas and spaces are ignored (i.e. first non–comma/non–zero character is displayed).
B1	ZRDMEM	Returns a value in the A register from RAM pointed to by the HL pair.
B2	ZRCPYU	Performs an LDIR instruction (switches ROM out first).
B3	ZRCPYD	Performs an LDDR instruction (switches ROM out first).

Function Number	Source Label	Description
B4	ZMOUT	Outputs a value held in the B register to the PSG port no. held in the C register.
B5	ZKSCAN	Returns the value, in the A register, of any key pressed. 00 is returned if no key is pressed. This is similar to MCAL 9B (ZRSCAN) except that it is unaffected by the key repeat speed. That is for each MCAL execution, a value is returned.
BC	ZZTIME	Set up CTC channels 2 to 3 to generate 1 second interrupts for clock.
BD	ZFDRST	Resets the FDC after an error. Also resets the PSG (using MCAL C0 (ZPINIT).
BE	ZSYSRS	This performs the following: 1. Clears the screen to 40 columns. 2. Resets all characters. 3. Removes sprites. 4. Resets the FDC and PSG. 5. Masks the keyboard, fire and ADC interrupts.

Function Number	Source Label	Description
BF	ZLOGO	Outputs '***EINSTEIN***' Logo.
C0	ZPINIT	Sets PSG register 7 to &7F and all other registers to 0.
C1	ZSREG	Sends an address held in the BC pair to the VDP. Data can then be output to VRAM from port 8. Subsequent data bytes sent will be loaded into subsequent VRAM locations. **Note**: A delay of 8 microseconds is necessary between any VRAM reads or writes (e.g. PUSH/POP). **Note**: When ROM is switched in, RST &20 will execute this MCAL.
C2	ZVRIN	Returns a value to the A register from the VRAM address pointed to by the BC pair.
C3	ZVROUT	Writes data held in the A regiter to the VRAM address held in the BC pair.

Function Number	Source Label	Description
CE	ZIMULT	Multiplies the contents of the DE pair and the contents of the BC pair and returns the value in DEHL (the DE pair is the most significant).
CF	ZPRM	Outputs a message to the screen. The data follows the CF data byte and must be a character in the range 0 to &7F. The message is terminated by adding &80 to the last character in the message.
D0	ZVOUT	Outputs a character from the A register to the current cursor position without incrementing the cursor position. This can be useful to prevent scrolling, linefeeds etc.
D1	ZSCURS	Returns VRAM addresses relating to the current cursor position. The ASCII text map address is returned in the BC pair (will be in the range &3C00 to &3FBF). The table pattern generator address (first byte) is returned in the DE pair (will be in the range 0 to &17FF). The start of the sprite pattern/text pattern table is returned in the HL pair (normally &1800).

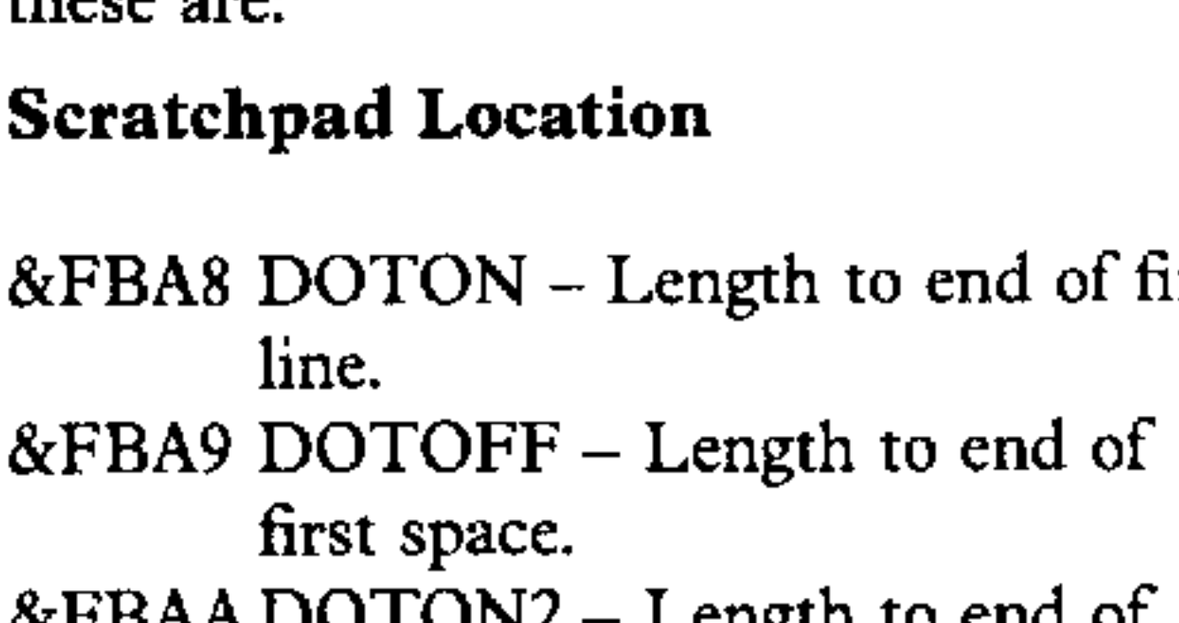
Function Number	Source Label	Description
D2	ZROM	Starts execution from address &4004 in the 2nd ROM. All registers can be used to pass and return values. Return is achieved by a RET instruction. **2nd ROM Protocol** 

Graphics Routines Stored in ROM

An understanding of these routines can be gained from using the graphics commands in XBAS. e.g. DRAW, FILL, ELLIPSE, POLYGON, PLOT.

Function Number	Source Label	Description
C4	ZPLOT	PLOTS or UNPLOTs a pixel. A = 1 for PLOT. A = 0 for UNPLOT. IX holds the x coordinate. IY holds the y coordinate.

....Contd.

Function Number	Source Label	Description
C5	ZPLTXY	Plots a point according to the line type (see below). IX holds the x coordinate. IY holds the y coordinate. **Line Type** Four scratchpad values contain information as to the line type to be drawn e.g. _____ ____ _ ____ _ ____ - these are: **Scratchpad Location** &FBA8 DOTON – Length to end of first line. &FBA9 DOTOFF – Length to end of first space. &FBAA DOTON2 – Length to end of second line. &FBAB DOTOF2 – Length to end of second space. 

Function Number	Source Label	Description
		Normally these 4 values are in ascending order. For a continuous line, DOTON is set to &FF and the other 3 to zero. For a continuous unplot, DOTOFF is set to &FF and the other 3 to zero. **Note**: If all line type bytes are zero, the system will hang up.
C6	ZPOINT	Returns the status of any pixel to the A register. 1 = foreground 0 = background BC contains the x coordinate. DE contains the y coordinate. The VRAM address of the pixel is returned in the BC pair.
C7	ZPNTXY	Returns the status of any pixel to the A register. This is identical to MCAL C6 except IX contains the x coordinate and IY contains the y coordinate. The VRAM address of the pixel is returnedin the BC pair.

Function Number	Source Label	Description
C8	ZDRWTO	This will draw a line from the coordinates held in the IX and IY registers to values in scratch pad locations &FB96 (X1) and &FB98 (Y1). Each is a two byte number. The type of line drawn is determined by the line type values (see MCAL C5).
C9	ZPOLYG	This draws a polygon (or ellipse). The polygon centre coordinates are held in scratchpad locations &FB9E (CX) and &FBA0 (CY), each is a two byte number. The horizontal and vertical radii are held in locations &FBA2 (RADX–2BYTE) and &FBA4 (RADY–2BYTE). The number of sides on the polygon is determined by a two byte number in scratchpad location &FBA6 (CINC). A value of 4 will give a circle. A value of &80 will give an octagon. A value of &100 will give a rectangle etc. The start angle is passed in the DE pair, the finish angle is passed in the BC pair each is in the range 0 to 1024. The lines drawn to the polygon centre are selected by setting the carry flag. The type of line drawn is determined by the line type values (see MCAL C5).

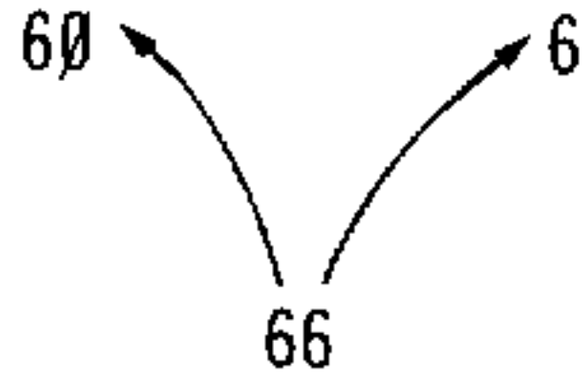
Function Number	Source Label	Description
CA	ZORGCO	Adds the x coordinate origin value held in scratchpad location &FB9A (ORGX – 2Bytes) to the contents of BC and returns the result in the BC pair and adds the y coordinate origin values held in scratchpad location &FB9C (ORGY – 2Byte) to the contents of DE and returns the result in the DE pair. This MCAL is of little use and is called from within other graphics MCALS. The values ORGX and ORGY are normally zero but when altered will cause all graphics output to be offset by that value. This is similar to the ORIGIN command in XBAS.
CB	ZCALAD	Returns the VRAM address in the BC pair for coordinates x and y passed in the IX and IY registers respectively. The pixel position within the 8 pixel row is returned in the E register, counting from the left hand pixel (0–7).
CC	ZSETCL	Writes data held in the A register to the pattern generator table address passed in the BC pair (0 to &17FF) and sets the corresponding byte in the pattern colour table (&2000 to &37FF) to the contents of scratch locations &FB39 (GCOLR).

Function Number	Source Label	Description
CD	ZFILL	Fills an area on screen surrounding coordinates passed in the IX and IY registers (x and y coordinates respectively). If the fill is in foreground (i.e. the point at x, y is not set) then scratch location &FBAD (FILLMOD) must be set to &FF. If the fill is in background (i.e. the point at x,y is set) then scratchpad location &FBAD (FILLMOD) must be set to zero. MCAL XPNTXY (C7) can be used to find the fill type needed.

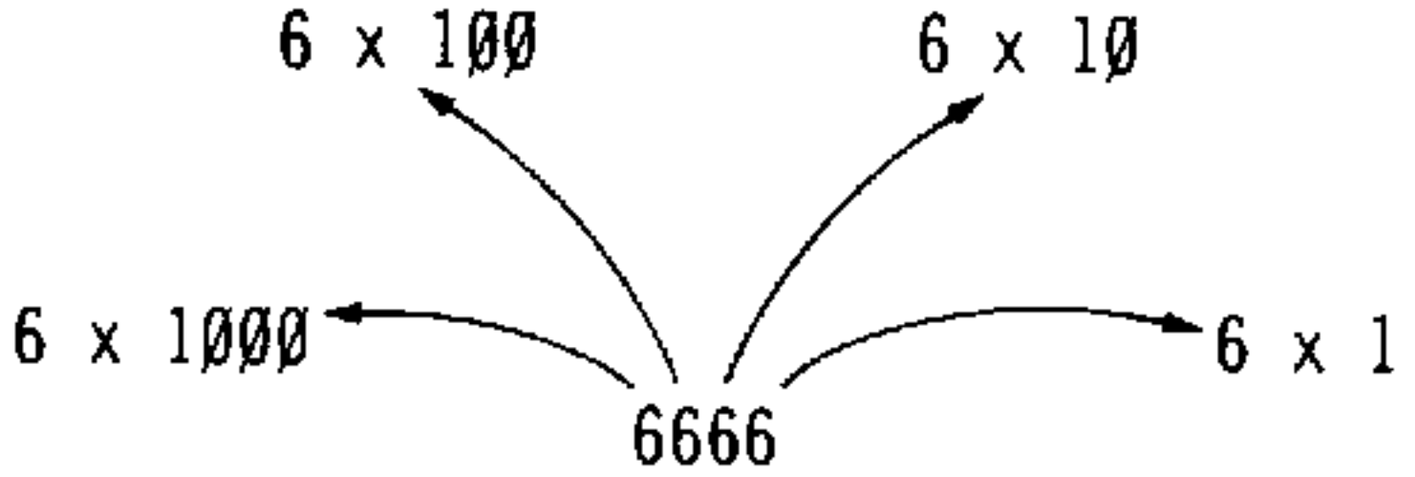
Appendix 5

Binary, Binary–Coded Decimal and Hexadecimal Notation

Modern counting systems, in general use throughout the world, use the decimal system. This has been developed to count past 9 and also below 1. In this system, the digits to the left in a number are of greater value than those to the right; for instance in the number 66 the first 6 has a value 10 times the second, e.g.

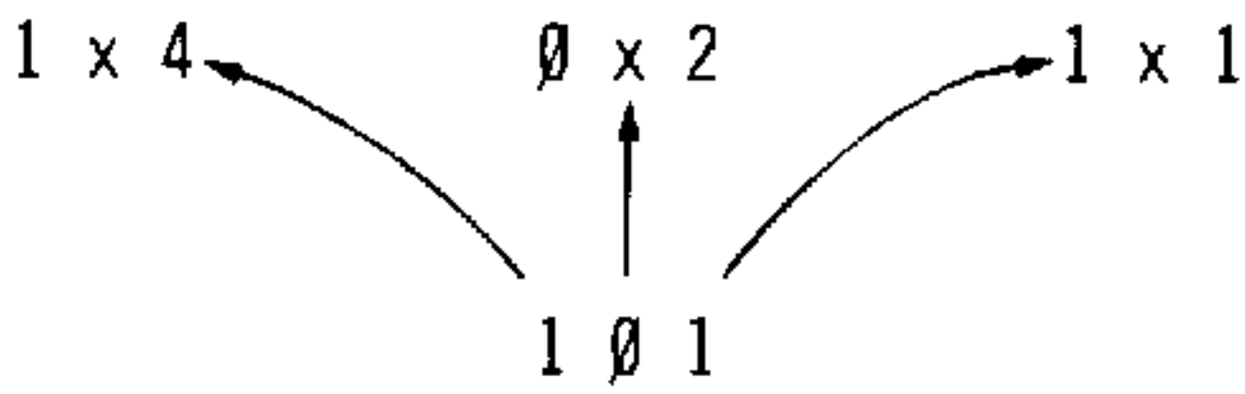


This is extended in larger numbers where digits to the left are successively greater by a multiple of ten, e.g.



A system where the position or place of a digit in a number affects its value is known as a '**place–value**' numbering system. In the decimal system, the values of digits increase in multiples of 10; this is known as the '**base**' for that system. Other systems use different bases but follow the same pattern as the decimal system, i.e. the place to the left is greater, being multiplied by the base.

The computer, being basically electrical in operation, can only recognise two states, on or off, often represented as '1' and '0'. Thus it uses the Binary system – i.e. base 2. Any number in binary consists simply of 0's and 1's, or electrically speaking, offs and ons (or electronically, zero volts and some volts). To count past one, the binary system must resort to place–value notation and, as with other systems, the multiplying factor is the base, i.e. 2. Thus, the number 101 in base 2 or 'binary' represents:



i.e. 4+0+1=5. Clearly the plethora of bases presents a problem when representing numbers. In decimal (base 10), '101' represents one hundred and one while in binary (base 2) '101' represents 5. To overcome this ambiguity, a convention exists when representing numbers so that the base is written to the right of the number, just below the line. Thus, the two numbers discussed above become:

101_{10} — One hundred and one in base ten.

101_{2} = Five in base two.

The present–day generation of home computers (1985–style) uses eight bit registers or memories and can, thus, represent numbers up to 11111111_2, i.e. 255 in base 10:

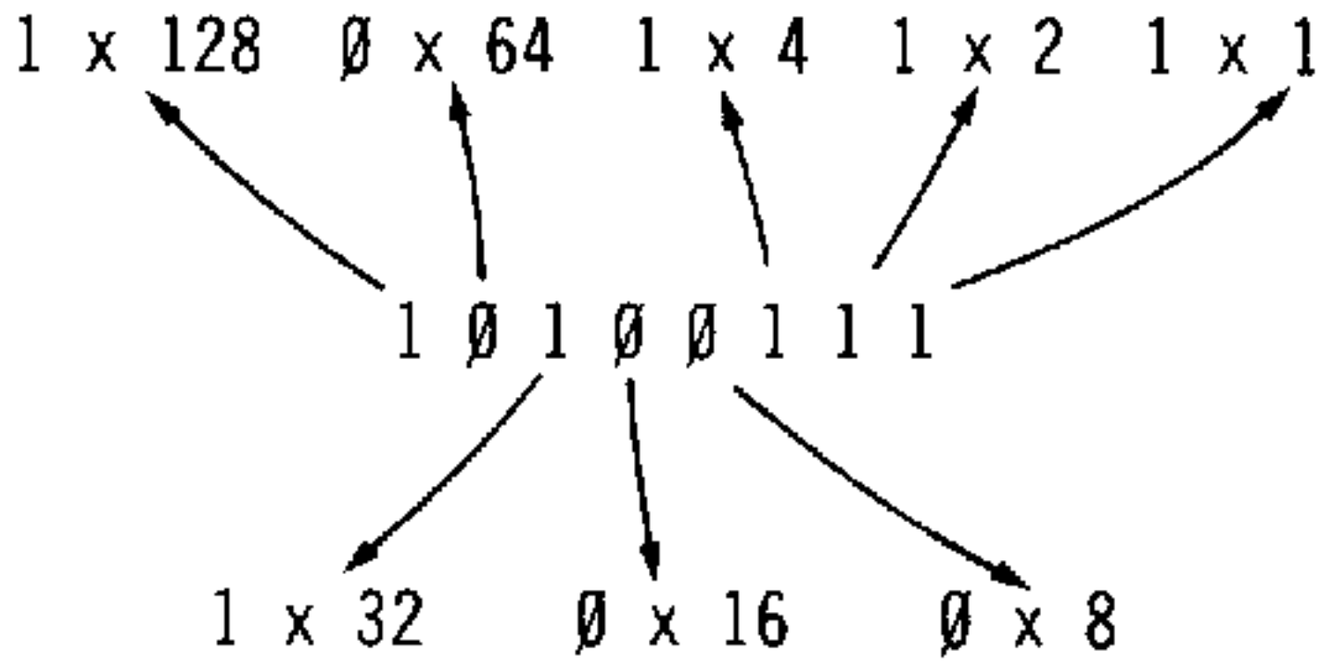
$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255_{10}$

1	1	1	1	1	1	1	1
128	64	32	16	8	4	2	1

Digit.
Equivalent
in base 10

FIGURE A5.1

By way of example, let's make another conversion – say, 10100111_2 to base 10.



Thus $10100111_2 = 1x128 + 0x64 + 1x32 + 0x16 + 0x8 + 1x4 + 1x2 + 1x1$
$= 128+32+4+2+1$
$= 167_{10}$

Just to check your understanding, have a go at the following:

EXERCISE A5.1

Calculate the value of the following binary numbers in base 10:

a) 00000011_2
b) 00000100_2
c) 10000000_2
d) 10000011_2
e) 10110111_2
f) 01110011_2

The answers are in the solutions Chapter.

To make idea of converting numbers from decimal to binary notation clearer try running the program 'TUTOR' on the disc.

The following display will appear: don't worry about the boxes marked 'HEX' or 'BCD', we will be coming to them a little later. What we are interested in at the moment are the 'DECimal' and 'BINary' boxes.

The flashing cursor indicates that the program is waiting for you do something. Helpfully, the cursor is next to the INCrement A command. Press 'ENTER'.

All the boxes relating to the A register have changed to display the value of 1 in the various different notations.

Keep pressing the ENTER key and as the values increase you will be able to see clearly how binary counts. You can use other commands by moving the cursor with the up/down arrow key. Using INC A or DEC A as appropriate, put '15' into the A register. (If you overshoot by one, note that the DEL key will un–do the last operation, whatever it was.) The binary box should contain '00001111'. Now increment this by one (move the cursor up to INC A using SHIFT and the up–arrow key). The left four bytes (the least significant bytes) have all changed to zero and the fifth byte has become one. One way of understanding this is to lay out the addition:

```
  1111
+    1
```

Adding the 1 to 1 gives '2' i.e. 0, carry 1. This carry then produces another '0' + a carry, and so on.

When the register is full, i.e. 11111111_2, the addition of a further 1 will clock the register back to zero and 256 will be lost. However, with the Z80 chip all is not really lost as the chip has a carry flag that stores the fact that a carry has occurred. (The carry flag is marked 'C' on the display.)

To see the carry flag work, increment A to 250 (a quick way to do this is to increment A up to say 50, then load B with the contents of A (LD B,A) and then ADD A,B a few times). It's now not too far to increment up to 255_{10} and 11111111_2.

Now load B with the contents of A (LD B,A) and INC B until it reaches 1. Now ADD A,B. Watch the binary box in the A register and the carry flag as you count past 255_{10} to 256_{10}. The binary cycles back to zero and the carry flag is set. This is a handy feature of the Z80 but must not be relied on as more than a temporary store of the carry. It's just as easily reset to zero as it is set to 1! Note that INC does not affect the carry flag, which is why ADD was used here.

To see if you really understand binary notation, try converting the following binary numbers to decimal on paper and then use the program to check your answers.

EXERCISE A5.2

a) 11111_2
b) 101001_2
c) 10111101_2
d) 10001000_2

The answers are in the solutions chapter.

Keep the TUTOR program in the computer; you will need it again soon!

While the 0's and 1's are convenient for the computer, they are much less so for the mere human so a compromise is sought. Decimal notation is of little use as, apart from 1_{10} and 1_2 there is no other correspondence. A further idea would be to take the whole eight binary bits as a digit (i.e. up to 255_{10}) and use a base of 256! What would you see as the objection to this? That's apart from the idea itself being a bit mind–bending!

Time to think ... The answer comes from an examination of the base 10 case in which ten digits (0 to 9) are needed to represent the ten steps up to 10. In the base 2 system, two digits are needed so base 256 would need 256 digits!

Hexadecimal

A compromise system adopted splits the eight bits up into two parts and represents these separately. Thus, the largest number to be represented is 1111_2 or 15_{10}. This requires, including the 0, sixteen different symbols. Those adopted for this job are:

Ø	1	2	3	4	5	6	7	8	9	1Ø	11	12	13	14	15
Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Decimal number

Symbol

FIGURE A5.2

Using this notation, any eight bit number can be represented by two symbols, one for the most significant four bits and one for the least significant four bits. To avoid this rather long description of these two halves of a byte, they are known as 'NYBBLES'. Thus a byte consists of two nybbles, a most significant nybble (MSN) and a least significant nybble (LSN) – see Figure A5.3.

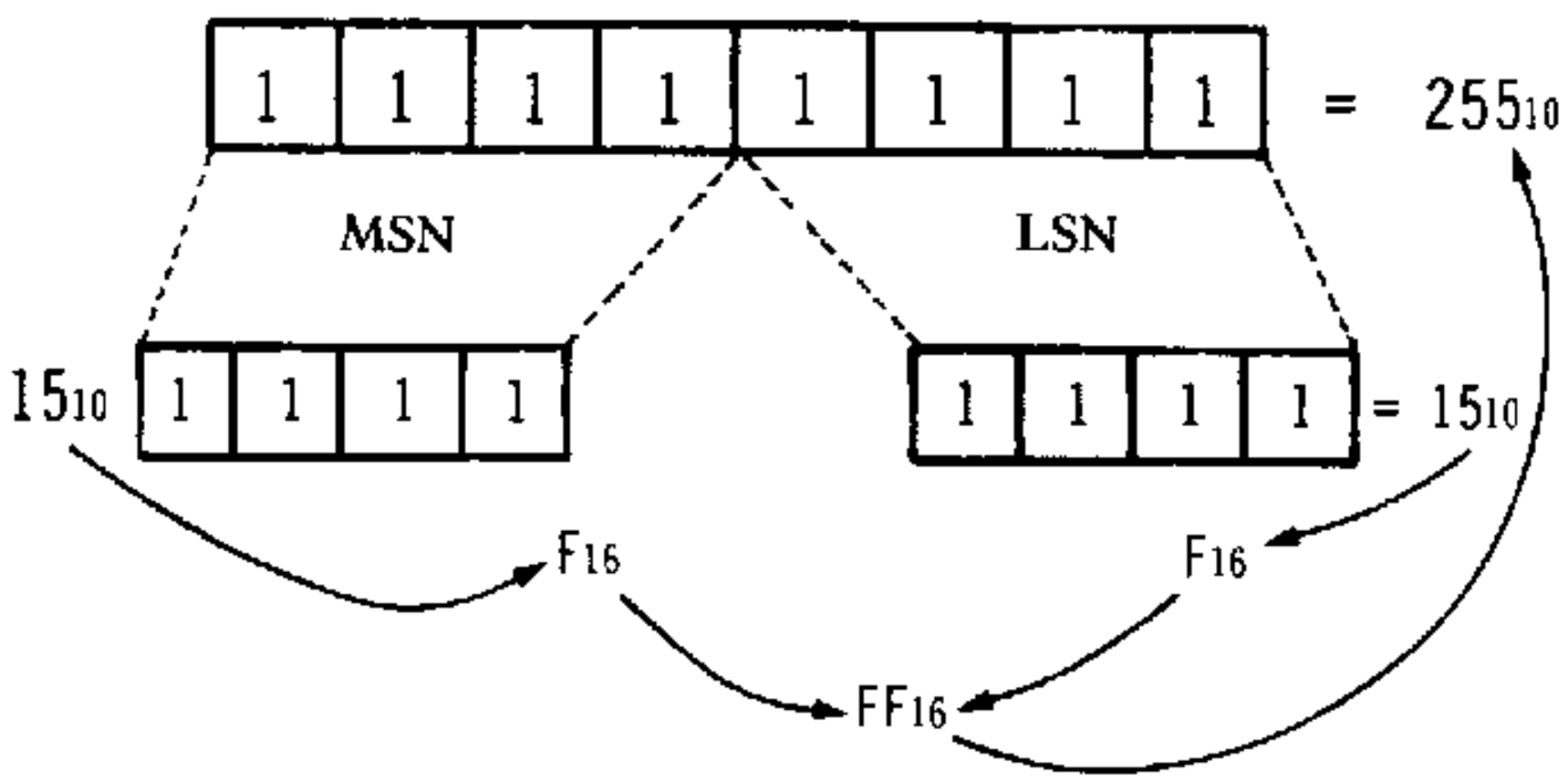


FIGURE A5.3

The system described, which uses sixteen symbols, is given the name HEXADECIMAL – usually abbreviated to HEX. Its major advantage, as far as computers are concerned, is that it is compatible with binary. Any eight bit binary number can be represented by two hexadecimal characters.

If you now look at the top right–hand boxes on the screen that we pretended not to notice earlier, you will see that one of them counts in HEX. The compatibility between binary and HEX shows wherever a major carry occurs – take for instance 1111_2, i.e. 15_{10} or F_{16}: one increment past this turns the binary ones to zeroes and adds a one to the left, i.e. to 10000_2 or 10_{16}. These major points of correspondence occur at:

$1_2 = 1_{16} = 1_{10}$

$0001\ 0000_2 = 10_{16} = 16_{10}$

$0000\ 0001\ 0000\ 0000_2 = 100_{16} = 256_{10}$

$0001\ 0000\ 0000\ 0000_2 = 1000_{16} = 4096_{10}$

As you will have already noticed, the hexadecimal numbers on the display are preceded by an '&' symbol. When entering hex numbers using the assembler, you should precede them with this symbol so the assembler will recognise them. (It might otherwise assume the numbers are decimal or even labels.)

Clear the A register (use XOR A). Then INC A a few times, watching the HEX and DECimal boxes. Up to 9, the hex characters coincide with the decimal ones and between 10 and 15 the single letters correspond to the decimal numbers. After 15, Hex to decimal conversion becomes a little more tricky, as the use of two numbers together, e.g. FF_{16}—255_{10}, once again calls for place–value notation. This time, as the base is 16 the ratio between any place and its neighbour is 16.

The values, in base 10, of the places in hexadecimal are:

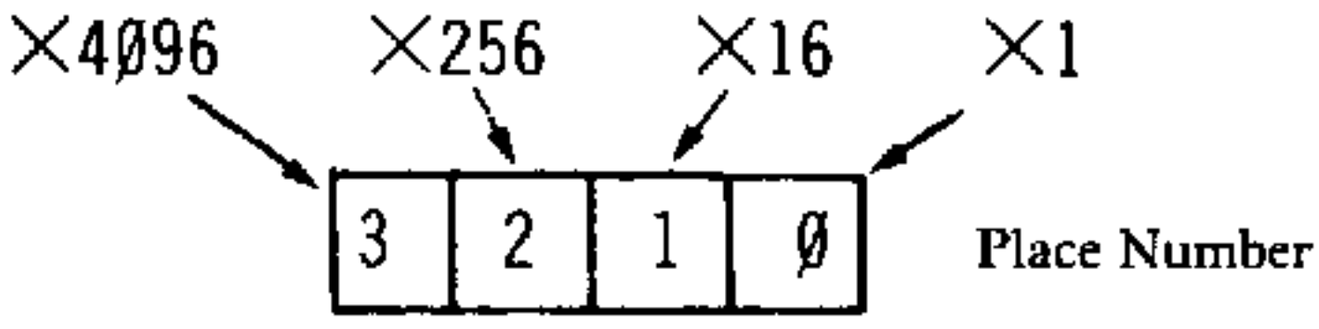


FIGURE A5.4

Using Figures A5.4 and A5.5, the breakdown of E92F in base 16, into 59695 in base 10, is illustated.

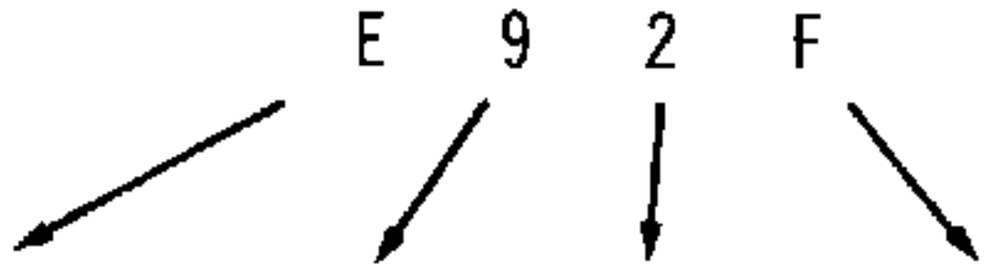


FIGURE A5.5

Now that hex is totally mastered (!) try the following; the first two are explained fully in the solutions chapter.

EXERCISE A5.3

Calculate the value in decimal of the following:

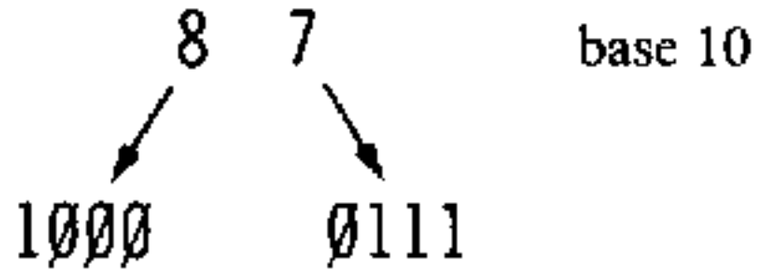
a) 09
b) 13
c) A5
d) AE
e) 0A
f) 1A
g) EA

Do the above exercise on paper first. You can check your answer to Exercise A5.3 using the program or look in the solutions chapter. Keep the program, you will need it in a moment.

Binary–Coded Decimal

As well as decimal, binary and hexadecimal notations, one other system is used in computing – Binary–Coded Decimal. As its name suggests, it is a hybrid form with elements from both binary and decimal. It is commonly used where an output is required in digital format, e.g. a digital clock, or when great precision is required and no bits can be dropped.

In BCD the normal decimal base is retained, i.e. one place is a factor of 10 times its neighbour but each individual digit is represented in binary. Thus the number 87_{10} would be represented:



i.e. $87_{BCD} = 1000\ 0111$ (or in binary 10000111_2)

FIGURE A5.6

As the largest digit required in decimal notation is 9, only four bits of binary are needed to represent this, i.e. $9_{10}=1001_2$, thus a BCD digit can be represented by a nybble and two digits by a byte. Figure A5.6 shows this, where 87_{10} is represented in BCD as 1000 0111. This can give rise to ambiguity in that 10000111 in binary is 135_{10}. To overcome this, BCD representations will be given the notation 10000111_{BCD}.

Using four bits of binary, it is possible to count up to 15_{10} (i.e. 1111_2—15_{10}) but in BCD the largest digit used is 9, so inevitably BCD is less economical in its use of space. Its largest digit, 9, is 1001_2 and when one is added to this it clocks over to 0000_{BCD} and carries the 1 to the next nybble, i.e.

$8_{10} = 0000\ 1000_{BCD}$
$9_{10} = 0000\ 1001_{BCD}$
$10_{10} = 0001\ 0000_{BCD}$
$11_{10} = 0001\ 0001_{BCD}$

FIGURE A5.7

The display we are concerned with this time is the one marked BCD, and the binary box. Change A to give it a start value of 1. If you increment through the first 9 counts you will see that up to 9_{10}, binary and BCD are identical (where the binary box contains the binary equivalent of the BCD number). However, as you increment from 9 to 10, keep an eye on the BCD box and you will see that the message 'NO BCD' appears. That is because 00001010_2

is not a valid representation of 10_{10} for BCD. The correct form would be 00010000_2. Increment A until it reaches this value. Suddenly the BCD box shows '10'. This example serves to emphasise the fact that you, the programmer, must be aware of which number system you are using, as 00010000_2 is the binary for 16_{10} as well as 10_{BCD}.

As the number in A increases, the uneconomical nature of BCD will become apparent. If you now increment A to 95_{BCD} and increment from there you will see the problem as 99_{BCD} changes to 100... Two nybbles cannot contain any number larger than 99 in BCD, hence the 'NO BCD' message will stay for all numbers greater than this. If you decrement back down to 99_{BCD} or less, the appropriate BCD value is again displayed.

As you know all about BCD now, try the following exercise!

EXERCISE A5.4

Convert the following decimal numbers into BCD:

a) 4
b) 10
c) 77
d) 97
e) 53
f) 102
g) 953
h) 2579

The answers are in the solutions chapter.

EXERCISE A5.5

Convert the following BCD numbers into decimal:

a) 0000 0001
b) 0000 1001
c) 0001 0101
d) 0010 0000
e) 0100 1001
f) 0010 0011
g) 1001 0111
h) 1000 1000

The answers are in the solutions chapter.

In the explanation given for the value of places in place–value notation a simplification was adopted in order to make these explanations clearer for the less mathematically inclined. However, if you wish to see a slightly more mathematical explanation, please read on. Otherwise – END OF APPENDIX 5.

With binary numbers, it was said that the places increase their value in multiples of 2, but the least significant bit of the binary number was equivalent to the same symbol base 10 (or for that matter base 3, or whatever). In actual fact, the multiplying factor is the base, raised to the power of its place, starting with zero at the left. E.g. in binary:

7	6	5	4	3	2	1	Ø	Place.
128	64	32	16	8	4	2	1	Multiplication factor
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Mathematically more precise factor.

Thus the least significant bit is multiplied by 2^0, i.e. 1. (If you are not sure of this try the direct program PRINT 2^0 in XBAS.) The next bit is multiplied by 2^1, and so on.

This rule holds good for ANY base. Let's apply it for hex, i.e. base 16:

Least significant bit factor $= 16^0 = 1$

2nd most significant bit factor $= 16^1 = 16$

3rd most significant bit factor $= 16^2 = 256$

Most significant bit factor $= 16^3 = 4096$

Appendix 6

The Assembler

This appendix explains the features of the assembler ('ASSEM')and of the disassembler ('DISSEM'). It is recommended that you read through it and refer to it whenever necessary.

Summary Of Commands

V	View program (list)
V n	View from line n
V n m	View from line n to line m
D n	Delete line n
D n m	Delete lines n to m
R	Renumber from 10 step 10
R n	Renumber from n step 10
R n m	Renumber from n step m
A	Automatic line numbers from 10 step 10
A n	Automatic line numbers from n step 10
A n m	Automatic line numbers from n step m
P 0	No display of program
P 1	Print to screen only
P 2	Print to printer only
P 3	Print to screen and printer
Z	Assemble into Z80 machine code
E	Execute program from location of ENT command

C &hhhh	Convert from hex to decimal
S name	Save source code 'name' to disc
L name	Load source code 'name'from disc
O &hhhh	set Offset for source code
W name	Write object code 'name' to disc

These commands will be explained in more detail on the following pages.

View Program

This command allows the whole source–code or a segment of it to be listed to the screen, as follows:

V	View source code program
V n	View from line n
V n m	View from line n to line m

Deleting Lines

Single lines can be deleted quite easily in just the same way as lines are deleted in BASIC, i.e. typing in the line number and pressing ENTER. You can, however, use the 'D' command as follows:

D n	Delete line n
D n m	Delete lines n to m

The latter form is the most useful of course. This allows a whole section of a program to be deleted in one go. Thus to delete lines 50 to 90 whilst leaving lines 10 to 40 and line 100 intact, the command

```
D 5Ø 9Ø
```

would be used. To delete a whole program, just make sure that the first and last line numbers given in the command cover the range of the program's line numbers. Thus to delete a program having line numbers 10–100, the command

```
D Ø 2ØØ
```

could be used.

Renumbering Programs

The R command is used to change the line numbers in all or part of a program:

R	Renumber the entire program from 10 step 10
R n	Renumber the entire program from n step 10
R n m	Renumber the entire program from n step m

The first option is useful for tidying up the line numbers in a program which has had many lines inserted and deleted.

The renumber command always renumbers the entire program. Thus a program having lines 10, 20, 35 and 50 would have line numbers 40,50,60,and 70 after an R 40, or 1, 3, 5 and 7 after an R 1 2.

Automatic Line Numbers

When typing in a program it can be tiresome to have a type in the line number at the beginning of each new line. The 'A' command will provide the next line number automatically whenever the ENTER key is pressed.

A	Auto line numbers from 10 in steps of 10
A n	Auto line numbers from n in steps of 10
A n m	Auto line numbers from n in steps of m

To terminate the function, a line containing just an '@' symbol (Shift zero) should be entered.

Printing Programs

Given that the appropriate printer interface and/or driver program is successfully installed/loaded prior to loading the assembler, the 'P' option allows programs to be output as follows:

P 0	No output, except immediate commands on screen
P 1	Print to screen only
P 2	Print to printer only
P 3	Print to screen and printer simultaneously

Having chosen say, P 3, a 'V' to list the program will cause it to list on both the screen and the printer. Thereafter, everything which appears on the screen will also appear on the printer. You should sure that the printer is 'on line' and switched on, as the computer will cease all operations until it is – i.e. the computer will appear to stop.

Assembling Programs

To assemble a source–code program into machine–code, just press 'Z' and then press 'ENTER'. If all goes well, the program will list on the screen showing both the source code and the machine code. If all is not well, an error mesage will appear telling you approximately what is wrong. To replace a faulty line in a program, just type in the appropriate line number and the correct instruction.

Execution of Programs

There are two main ways of telling the computer to run an assembled machine–code program.

If the assembler directive ENT has been used then a simple 'E' will cause the program to Execute (i.e. run) from *within* the assembler starting with the instruction immediately following the ENT.

If the program was assembled containing the ORG directive, e.g. 'ORG 256', and no ENT, then it can be run from DOS as done throughout this book. To do this, an offset should be set (see Offset), then the program assembled with an ORG as its first line. Then the assembled object code should be saved to disc with the 'W name' command and the assembler exited using 'Q' (for Quit). To run the program, simply type in its name. This will cause the program to execute from address 256, as given in the ORG directive. (See also the section dealing with assembler directives later in this appendix.)

Converting from Hex to Decimal

To convert a number from hexadecimal to decimal, the command 'C &number' can be used for numbers from 0 to &FFFF (0 to 65535 decimal).

Saving Source Programs To Disc

The S option will save source–code to disc, under the given name:

```
S name.ccc
```

The name may be from one to eight characters long followed by a dot and another one to three characters. If the dot and one to three characters are omitted, the program will be saved as 'name.ASM'

Loading Source Programs from Disc

To load a source–code program, the command should be of the form

```
L name.ccc
```

The dot and final three characters may be omitted if you wish to load a 'name.ASM' file.

Setting an Offset

Normally, programs will be written so as to execute from location 256. However, when writing the program, the assembler occupies locations 256 to about 11008 (&100 to &2B00). Thus, rather than assembling the object code to its run address (set by the ORG directive), it will be assembled to the address chosen by the ORG address *plus* the Offset number – although the assembler will display the actual run–time addresses as it assembles the program. When the program is later run from DOS, it will automatically run from just the ORG address. The form of the Offset command is:

```
O &hhhh
```

If ENT is being used, no offset should be set (i.e. O &0).

Writing Object Code to Disc

This command allows you to save assembled code to disc. The command takes the form

```
W name
```

Files will be saved as 'name.COM' and can be run direct from DOS – i.e. simply by quitting the assembler (Q) and then typing in their 'name'.

Assembler Directives

As well as accepting the full Z80 instruction set, the assembler will accept certain assembler directives. An assembler directive is an instruction to the assembler instead of the Z80 (they are sometimes called pseudo–opcodes for this reason).

Assembly directives considerably increase the legibility and ease of writing programs.

ENT	Causes the object code to be placed after the source code, i.e. at &C000, in a form suitable for running using the assmbler's execute program option (E).

Note that no offset should be set if ENT is to be used.

When writing programs that require more storage for data than is available from the registers or easily from the stack, it is necessary to use memory locations to store this data. Ideally these memory locations would be situated in the same vicinity as the program. As the exact length of the program is not usually known, calculating the address of a memory location situated at the end of the program is not easy. The solution to the problem is to use the following assembler directive.

DEFS n	Reserves **n** bytes of memory starting at the current value of the location counter.

To allow easy reference to this storage area, a label is assigned to this directive. For example to sent up a four–byte buffer referenced by the label 'STORE:', the following line would be used

```
STORE:  DEFS 4
```

When the assembler encounters this command it will reserve four bytes of memory and assign to the label 'STORE:' the address of the first byte in this buffer.

For example:

Suppose that an 8 by 8 bit multiplication program produces a 16 bit result in HL; if this result will be needed further on in the program it will have to be kept.

By using the following program the result can be obtained by referencing the label STORE:

```
STORE:   DEFS 2
            .
            .
            .
Multiplication program
            .
            .
            .
         LD (STORE:),HL
            .
            .
            .
```

Thus the contents of the memory location addressed by STORE contain the result of the multiplication.

Note that you should be careful that your program won't try to execute the stored bytes – either by jumping around them or by putting them after the final RET.

As well as creating buffers, assembler directives can be used for many other functions. The first to be considered is assigning numeric values to labels.

EQU nn	Assigns the value **nn** to the preceding label.

This form of directive is used mainly for setting up constants.

For example:

```
LOOP:    EQU 20
         LD A,LOOP:
```

When assembled and executed, this would load A with 20.

The following directive lets the assembler know that the following **program** is to be called from DOS and executed from address nn.

ORG nn	Set the location counter to **nn**

This feature of the asembler allows programs to be assembled so as to reside anywhere in memory at run–time. Note: the value nn will normally be 256_{10} or greater. If you type in an invalid location the assembler will inform you. Generally, an ORG and an Offset will be used together when writing a program, as used throughout this book.

Many programs require specific memory locations to be loaded with data. There are three directives to accomplish this; the first two load numeric data:

DEFB n	**DEF**ine **B**yte: Stores **n** at the current value of the location counter.

DEFW nn	**DEF**ine **W**ord: Stores the LSB of the 2 byte word **nn** at the current value of the location counter and the MSB at the current value of the loction counter + 1.

The last directive stores the ASCII representation of a string in memory. This is very useful for printing messages on the screen.

DEFM "s"	**DEF**ine **M**essage: Set the contents of memory, starting with the current value of the location counter, to the ASCII representation of string **"s"**.

Try the following program (set the offset to zero with 'O &0' and don't use ORG):

```
        ENT
        LD B,10
        LD HL,MESSG:
LOOP:   LD A,(HL)
        RST 8
        DEFB &9E
        INC HL
        DJNZ LOOP:
        RET
MESSG:  DEFM "A MESSAGE!"
```

Assemble the program as usual with 'Z' and then execute it by pressing 'E' followed by ENTER. If you wanted the message to end with return, a DEFB 13 could be put on the line after the DEFM, and B loaded with 11 instead of 10.

Addition of Operands and Direct Loading of ASCII Characters

In order to print single charcters on the screen it is not necessary to know their ASCII values. An easier and more direct method is to type the required character bewteen quotes. The assembler then calculates the required ASCII code.

Thus to load the ASCII representation of A into the Accumulator it is possible to use the following line:

```
LD A,"A"
```

This will load A with 65. Likewise, to load A with "B" you could use:

```
LD A,"A"+1
```

These functions are very useful at times, especially for setting up arrays of data.

Comments

The assembler will allow comment lines to be included in the text provided that the first character in that line is a semi–colon. Thus a sample program containing comments would appear as below:

```
        ENT
        LD A,65
;LOAD A WITH "A"
        RST 8
        DEFB &9E
```

Note that comments must be on separate lines from instructions; the two cannot be combined on one line.

The Disassembler

On the disc which comes with this book is a program called 'DISSEM'. This is a disassembler – i.e. a program to take machine–code and convert it into assembly language. It is loaded from DOS in the same way as other programs; i.e. just type in its name and press ENTER.

Suppose you have the assembled the following program and saved the object code to the disc under the name 'EXAMPLE.COM'.

```
        ORG 256
        LD A,65
        RST 8
        DEFB 158
        RET
```

Now suppose you wish to disassemble that saved object code to find out what the source code was (i.e. to return to the original program shown above). Make sure a disc containing both DISSEM and EXAMPLE.COM is in the drive, then type in

```
DISSEM EXAMPLE.COM
```

On the screen the following message should appear:

```
Disassemble to:
```

Here, it is asking you how much code you want disassembled – i.e. the last location we wish it to disassemble to. As we know that this is a short program, and that it starts at location 256, we can enter a number like 300. So, enter

```
300
```

Were you to simply press ENTER instead, it would automatically disassemble to the end of the first 128 bytes, or to the end of the first 256 bytes if EXAMPLE.COM was saved from DOS.

The next message to appear on the screen is

```
Data area A from:
```

We will ignore this for now, so just press ENTER. The next message to appear is:

```
Constants to be shown in
decimal or hex (D or H)?
```

Press D or H depending on whether you prefer decimal or hex numbers – do **not** press ENTER. This example will assume you've pressed D.

On the screen should appear:

```
Characters corresponding to
code to be shown (Y or N)?
```

Press Y or N depending on whether you would like the ASCII equivalents of code to be displayed or not. This example assumes you've pressed N.

The disc drive will whirr as the code is disassembled and saved to disc as 'EXAMPLE.DSM'. You will not see the code appearing on the screen.

To see the disassembled code, load ASSEM then load EXAMPLE.DSM with the L option. You can view the program with the V option. As you will see, too much code has been disassembled. Use the D option to delete lines 261 to 300

Now view the program. It should look like this:

```
256          LD   A,65
258          RST  &8
259          SBC  A,(HL)
260          RET
```

As you can see, this is wrong. The line which says 'SBC A,(HL)' should in fact say 'DEFB 158'. The reason the disassembler got it wrong is that it is not possible for a disassembler to tell the difference between a byte of data and the op–code for an instruction. That is what the 'data area' prompt was for earlier. Try it now: quit the assembler with the Q option, then type 'DISSEM EXAMPLE.COM'.

Disassemble to 260, then enter data area A from 259 to 259 – i.e. one byte. As there is only one data area, just press ENTER for data area B (you can enter a maximum of 26 data areas should you wish to). The answers to the next two questions are up to you.

When the '0:' prompt reappears, load ASSEM, then load EXAMPLE.DSM and view it. This time, the DEFB is present.

If you had chosen the ASCII equivalents option as Yes, then a comment line would be included between the program lines with constants shown as ASCII values; e.g. 65 would be shown as A, etc. The program 'STRIP' will remove them, saving the file as, for example, EXAMPLE.DSM.

Copying the Assembler

To copy the assembler to another disc, simply load it from DOS with the instruction

```
LOAD ASSEM.COM
```

The screen should come up with a message saying how large the program is, e.g.

```
44 Block(s)
```

To save this file onto another disc, use

```
SAVE 44 ASSEM.COM
```

A similar procedure can be used for DISSEM and other files.

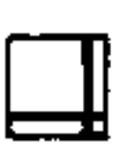
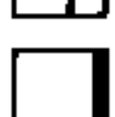
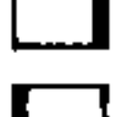
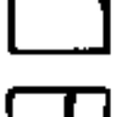
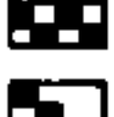
Appendix 7

ASCII CODES

Code	Symbol	Code	Symbol	Code	Symbol
	NUL	16	DLE	32	SP
1	SCREEN DUMP	17	CURSOR ON	33	!
2	STX	18	PRINTER ON	34	"
3	ETX	19	PRINTER OFF	35	#
4	CURSOR RIGHT	20	CURSOR OFF	36	$
5	ENQ	21	DELETE TO END OF LINE	37	%
6	DELETE CHAR FROM RIGHT	22	ERASE TO END OF SCREEN	38	&
7	BEEP	23	ETB	39	'
8	FE,(BS)	24	DELETE WHOLE LINE	40	(
9	FE,(HT)	25	DEL	41	)
10	FE,(LF)	26	INS	42	*
11	FE,(VT)	27	ESC	43	+
12	FE,(FF)	28	FS	44	,
13	FE,(CR)	29	GS	45	-
14	SO	30	HOME	46	.
15	SI	31	US	47	/

Code	Symbol	Code	Symbol	Code	Symbol
48	0	69	E	90	Z
49	1	70	F	91	←
50	2	71	G	92	½
51	3	72	H	93	→
52	4	73	I	94	↑
53	5	74	J	95	
54	6	75	K	96	£
55	7	76	L	97	a
56	8	77	M	98	b
57	9	78	N	99	c
58	:	79	O	100	d
59	;	80	P	101	e
60	<	81	Q	102	f
61	—	82	R	103	g
62	>	83	S	104	h
63	?	84	T	105	ı
64	@	85	U	106	j
65	A	86	V	107	k
66	B	87	W	108	l
67	C	88	X	109	m
68	D	89	Y	110	n

Code	Symbol	Code	Symbol	Code	Symbol
111	o	132	f4	153	
112	p	133	✢/f5 (a/w)	154	
113	q	134	f6	155	Fc
114	r	135	bl/f7	156	
115	s	136	bs	157	
116	t	137	ht	158	
117	u	138	lf	159	
118	v	139	vt	160	SP
119	w	140	ff	161	
120	x	141	cr	162	
121	y	142		163	
122	z	143		164	
123	¼	144	dl	165	
124		145		166	
125	¾	146		167	
126	÷	147		168	
127	■	148		169	
128	f0	149		170	
129	⇨/fi	150		171	
130	f2	151		172	
131	f3	152		173	

Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
174		195		216		237	
175		196		217		238	
176		197		218		239	
177		198		219		240	
178		199		220		241	
179		200		221		242	
180		201		222		243	
181		202		223		244	
182		203		224		245	
183		204		225		246	
184		205		226		247	
185		206		227		248	
186		207		228		249	
187		208		229		250	
188		209		230		251	
189		210		231		252	
190		211		232		253	
191		212		233		254	
192		213		234		255	DEL
193		214		235			
194		215		236			

Solutions

Solutions Chapter

Chapter 1

EXERCISE 1.1

```
ORG 256
LD A,65
RST 8
DEFB &9E
RET
```

EXERCISE 1.2

```
ORG 256
LD A,7Ø
RST 8
DEFB &9E
LD A,82
RST 8
DEFB &9E
LD A,69
RST 8
DEFB &9E
LD A,68
RST 8
DEFB &9E
RET
```

Chapter 2

EXERCISE 2.1

```
        ORG 256
        LD B,1Ø
        LD A,65
        RST 8
        DEFB 158
        DEC B
        JR NZ,26Ø
        RET
```

EXERCISE 2.2

The JR NZ instruction is used in preference to the JP NZ instruction because the jump distance is within +129 and –126 bytes and the JR NZ instruction requires less time and memory to execute. It is therefore more efficient.

EXERCISE 2.3

```
        ORG 256
        LD C,26
        LD A,65
NXT:    RST 8
        DEFB 158
        INC A
        DEC C
        JR NZ,NXT:
        RET
```

Chapter 3

EXERCISE 3.1

Memory location	Contents
:	:
:	:
200	E
201	D
202	L
203	H
:	:
:	:

EXERCISE 3.2

```
ORG 256
LD A,255
LD (64424),A
LD IX,0
LD IY,0
LD A,100
LD (64412),A
LD A,200
LD (64414),A
LD A,0
LD (64425),A
LD (64426),A
LD (64427),A
RST 8
DEFB &C8
RET
```

EXERCISE 3.3

```
          ORG 256
START:    LD A,255
          LD (64424),A
          LD A,Ø
          LD (64425),A
          LD (64426),A
          LD (64427),A
LINE1:    LD IX,1ØØ
          LD IY,2ØØ
          LD A,15Ø
          LD (64412),A
          LD A,16Ø
          LD (64414),A
          CALL DRAW:
          LD (64415),A
          CALL DRAW:
LINE2:    LD IX,15Ø
          LD IY,16Ø
          LD A,Ø
          LD (64412),A
          LD (64414),A
          CALL DRAW:
          RET
DRAW:     RST 8
          DEFB &C8
          RET
```

Note: the labels START, LINE and LINE2 serve no purpose other than to indicate what each section of the program is doing.

EXERCISE 3.4

```
        ORG 256
        LD BC,35ØØØ
        LD A,65
        LD E,3
NXT:    LD (BC),A
        INC BC
        DEC E
        JR NZ,NXT:
```

Program 3.3

```
        LD E,3          No. of times to loop.
        LD BC,35ØØØ     Load start location of 'A's.
PUT:    LD A,(BC)       Put 'A' in accumulator.
        INC BC          Add 1 to data location.
        CALL PRINT:     Put 'A' on screen.
        DEC E           Decrement loop number.
        JR NZ,PUT:      If loop number<>0 then loop.
        RET             End.
PRINT:  RST 8
        DEFB &9E
        RET
```

EXERCISE 3.5

```
        ORG 256
        LD BC,35ØØØ
        LD A,65         ASCII 'A'.
        LD E,26         26 loops.
NXT:    LD (BC),A
        INC BC
        INC A
        DEC E
        JR NZ,NXT:
RESET:  LD E,26         Reset pointers.
```

```
        LD BC,35ØØØ
PUT:    LD A,(BC)
        INC BC
        CALL PRINT:
        DEC E
        JR NZ,PUT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

EXERCISE 3.6

```
        ORG 256
        LD IX,35ØØØ
        LD A,83
        LD (IX+Ø),A
        LD (IX+2),A
        INC A
        INC A
        LD (IX+1),A
        LD A,65
        LD (IX+3),A
        LD A,78
        LD (IX+4),A
        LD A,(IX+2)
        CALL PRINT:
        LD A,(IX+3)
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

Chapter 4

EXERCISE 4.1

```
        LD A,2ØØ
        ADD A,48
        RST 8
        DEFB &9E
        RET
```

This prints a wavy line graphics character.

EXERCISE 4.2

```
        LD A,&41
        ADD A,&1Ø
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

This places a 'Q' on the screen.

EXERCISE 4.3

```
LD C,&FA          LSB of 250
LD A,&58          LSB of 600
AND A
ADD A,C
ADD A,65
LD (&7ØØØ),A
LD C,Ø            MSB of 250
```

```
        LD A,2                          MSB of 600
        ADC A,C
        ADD A,65
        LD (&7ØØ1),A
        LD A,(&7ØØØ)
        CALL PRINT:
        LD A,(&7ØØ1)
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

This produces a subscript F, a superscript P and a C.

EXERCISE 4.4

```
        LD C,9
        LD A,233
        SUB C
        RST 8
        DEFB &9E
        RET
```

It is not necessary to subtract 65 from the answer as character 224 is a printable character (three short diagonal lines).

EXERCISE 4.5

```
LD A,126
ADD A,97
LD C,153
SUB C
RST 8
DEFB &9E
RET
```

This puts 'F' on the screen.

EXERCISE 4.6

```
LD DE,4ØØ8
LD HL,4248
AND A
SBC HL,DE
LD A,L
RST 8
DEFB &9E
RET
```

This puts two horizontal bars on the screen.

EXERCISE 4.7

```
LD HL,35ØØØ
LD A,1Ø
LD (HL),A
INC HL
LD A,2Ø
LD (HL),A
```

```
        LD A,65
        ADD A,(HL)
        LD (HL),A
        DEC HL
        LD A,65
        ADD A,(HL)
        LD (HL),A
        CALL PRINT:
        INC HL
        LD A,(HL)
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

This puts 'KU' on the screen.

Chapter 5

EXERCISE 5.1

```
        ORG 256
        LD A,7
        ADD A,12
        DAA
        ADD A,65
        RST 8
        DEFB &9E
        RET
```

This prints a 'G' on the screen.

EXERCISE 5.2

```
ORG 256
LD A,&35
SUB &12
DAA
ADD A,65
RST 8
DEFB &9E
RET
```

This puts a lower case 'd' on the screen, corresponding to the BCD representation of &35–&12 (53–18 in decimal).

EXERCISE 5.3

1. C=Ø
2. C=1
3. C=Ø

EXERCISE 5.4

ØØØØØ11

EXERCISE 5.5

73

EXERCISE 5.6

1. 11Ø1
2. 251
3. 17

EXERCISE 5.7

```
        ORG 256
        LD A,11
        LD B,116
        XOR B
        CALL PRINT:
        LD A,8
        LD B,216
        XOR B
        CALL PRINT:
        LD A,25
        LD B,255
        OR B
        LD B,2ØØ
        AND B
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

This should produce characters 203, 208 and 200, i.e. a small chequerboard, vertical bars, and a half–cursor block respectively.

EXERCISE 5.8

1. Ø1ØØ
2. Ø1ØØØ1Ø
3 1 (14=1110, 1's comp=ØØØ1)

EXERCISE 5.9

```
1. -3    1101
   10    1010
    7    0111+carry

2. -1    1111
    7    0111
    6    0110+carry

3. -10   0110
    8    1000
   -2    1110
```

EXERCISE 5.10

```
        ORG 256
        LD A,20
        CPL
        INC A
        ADD A,98
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

This puts an 'N' on the screen.

Chapter 6

EXERCISE 6.1

The solution is the same as Program 6.1 except for these lines.

```
LD C,12
LD E,5
```

The program should put character 125 on the screen, i.e. a ¾ symbol.

EXERCISE 6.2

```
        ORG 256
        LD C,124
        LD E,146
        LD D,Ø
        LD B,8
        LD HL,Ø
NXTB:   SRL C
        JR NC,NOADD:
        ADD HL,DE
NOADD:  SLA E
        RL D
        DEC B
        JR NZ,NXTB:
        LD A,H
        CALL PRINT:
        LD A,L
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

This gives 'F'

EXERCISE 6.3

The easy way to do this is to delete DEC B and replace the line JR NZ,NXTB: with:

```
DJNZ NXTB·
```

EXERCISE 6.4

1.
```
        ORG 256
        LD A,5
        RLCA
        RLCA
        RLCA
        RLCA
        RLCA
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

2.
```
        ORG 256
        LD A,254
        RRCA
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

EXERCISE 6.5

```
        ORG 256
        LD A,255
        CALL PRINT:
        RES 4,A
        CALL PRINT:
        SET 4,A
        RES 3,A
        CALL PRINT:
        RET
PRINT:  RST 8
        DEFB
        RET
```

Chapter 7

EXERCISE 7.1

```
        ORG 256
        LD B,"B"
        LD C,"C"
        LD D,"D"
        LD E,"E"
PRNT1:  LD A,B
        CALL PRINT:
        LD A,C
        CALL PRINT:
        LD A,D
        CALL PRINT:
        LD A,E
        CALL PRINT:
PRNT2:  PUSH BC
```

```
        PUSH DE
        POP BC
        POP DE
        LD A,B
        CALL PRINT:
        LD A,C
        CALL PRINT.
        LD A,D
        CALL PRINT:
        LD A,E
        CALL PRINT·
        RET
PRINT:  RST 8
        DEFB &9E
        RET
```

Appendix 5

EXERCISE A5.1

a) 3
b) 4
c) 128
d) 131
e) 183
f) 115

EXERCISE A5.2

a) 31
b) 41
c) 189
d) 136

EXERCISE A5.3

a) 9
b) 19
c) 165
d) 174
e) 14
f) 26
g) 234

EXERCISE A5.4

a) 0000 0100
b) 0001 0000
c) 0111 0111
d) 1001 0111
e) 0101 0011
f) Too big for two–byte BCD
g) Too big for two–byte BCD
h) Too big for two–byte BCD

EXERCISE A5.5

a) 1
b) 9
c) 15
d) 20
e) 49
f) 23
g) 97
h) 88

Glossary

ADDRESS

The computer has lots of chips in it, in which to remember data. It stores up data in little groups of 8 bits, called bytes. The computer has lots of these bytes in its memory, we call each of the places that holds a byte a memory location. Each memory location has a special number associated with it, just like a house number; this number is called the 'address' of that memory location. It helps the computer find the byte it wants.

ALGORITHM

An algorithm is a set of instructions for doing a particular job. Cooking recipes are algorithms and so are flow charts and computer programs.

ARCHITECTURE

The arrangement or design of the logic in a computer system.

ARITHMETIC AND LOGIC UNIT or ALU

The area in the central processor that does all the arithmetic and logical operations.

ASCII

An abbreviation for American Standard Code for Information Interchange. In the computer, information is stored in the form of binary numbers so, in order to represent a letter in the computer, it was decided that each letter of the alphabet should be given a special number or code. This code is called the

ASCII code of the letter. Not all the ASCII codes represent letters, some represent other characters like =,+,!,? or /. Also there are other codes that represent new lines etc. Many computers do not strictly conform to the ASCII standard but most conform to a large extent.

ASSEMBLER

A program which translates assembly language into machine code.

ASSEMBLER DIRECTIVE

These commands tell the assembler something, e.g. ORG. This instructs the assembler to start storing the object code at the given address. See also Assembler.

ASSEMBLY LANGUAGE

A 'low level' computer language, but not as 'low level' as machine code. Assembly language is a language designed to make it easier to enter machine code programs into a computer. A program called an 'assembler' translates assembly language programs ('source code')directly into machine code ('object code').

BASIC

Beginners All Purpose Symbolic Instruction code. A 'high level' programming language.

BIT

One digit, either 0 or 1, in a binary number.

BYTE

A group of 8 bits used to represent a number between 0 and 255.

CENTRAL PROCESSOR UNIT

The part (in this case the Z80 chip) of the computer that does all the executing of machine code programs (every program boils down to machine code of one form or another!).

COMMANDS

The near English equivalents of machine code. We actually write commands when entering assembly language. Thus 'LD A,B' is a command; 80 is the machine code equivalent. See also 'mnemonics'. Another word used for command is 'instruction'.

COMPILE

To create the object programs from the source.

COMPILER

A program which converts a whole program written in a 'high level' language such as BASIC into machine code before the program is run. Speeds comparable with assembly language/machine code programs can be achieved. See also 'Assembler', 'Disassembler' and 'Interpreter'.

COMPLEMENT

A 'reverse' form of a binary number in which all the 1's are swapped for 0's and all the 0's for 1's, e.g.

```
10111100 becomes 01000011
```

CONTROL CHARACTERS

Control charaters do not display anything but serve to perform some action such as 'new line' <RETURN> or clear screen.

CRASH

If the Z80 is told to execute a program in memory that is incomplete in some respect then it may end up executing what it should not be executing! The Z80 is not very clever and can easily attempt to execute the contents of a buffer, string variable or any other kind of data. When this happens the Z80 gets in a mess, jumping around, executing a bit here and a bit there. It may store things all over the place, generally making a mess of what you have in memory. You will need to reset the computer to bring the Z80 to its senses. (A system crash will not do physical damage to the system. Crash the system for fun if you want!)

CRYSTAL OSCILLATOR

Device based on the vibration of a quartz crystal to produce an oscillating electrical voltage. Used in electronic watches, hence quartz watch, also on timing devices (clocks) for some computer systems.

DATA BUS

A group of electrical pathways used by a computer system to communicate internally.

DOUBLE PRECISION

A rather general term meaning twice as precise. Used in this book to mean mathematics done on numbers 2 bytes long (i.e. ranging from 0 to 65535) rather than one byte long.

DISASSEMBLER

A program which translates machine code into assembly language.

EXECUTE

Used, in the context of this book, to mean the running of a machine–code program.

FLAG

A special location, usually just one bit, that is set to a particular value if some condition was true or to some other value if it was not. It could be altered on the result say, of a compare operation between two numbers, if they were equal it would be set to 1 and they were not it would be set to 0.

GATE

Generally, an electronic device for performing logical operation. Voltage levels are used to indicate the values TRUE and FALSE, or 1 and 0. (Pneumatic logic gates also exist, but not in this computer!).

INDIRECTION

The process of looking up the address to use in an instruction, when the address is not actually given in the instruction: The instruction contains the address at which the address to use is stored.

INSTRUCTIONS

The near English equivalents of machine code. We actually write instructions when entering assembly language. Thus, ADD A,B is an instruction; 80 is the machine code equivalent. See also 'mnemonics'. Another word used for instruction is 'command'.

INTERFACE

A device to allow two electronic systems, not normally compatable directly, to communicate with each other.

I/O

Abbreviation for Input/Output

INTERPRETER

A program which reads another program (eg. a BASIC program) and converts it instruction by instruction into machine code which is then immediately executed. Note that this is different from an assembler or a compiler which convert whole programs (the 'source code'). Programs in an 'interpreted' language such as BASIC run more slowly than assembly language and compiled programs as each BASIC instruction has to be interpreted every time it is executed, whereas, say, an assembly language program only has to be converted into machine code once before the program is run. See also 'Compiler'.

LOAD

To load a register or the accumulator means to put some number into it. The number can be obtained in various ways.

LOGICAL OPERATOR

Operator or function that only works with the values TRUE and FALSE. The most basic logical operators are AND, OR and NOT. NOT just returns the complement (binary opposite) of its input.

LOW and HIGH LEVEL LANGUAGES

The 'level' of a computer language refers roughly to its similarity to English or other spoken languages. The higher the level, the closer the language is to a proper spoken language. The lower the level the closer the language is to the inscrutable numbers of machine code. High level languages are characterised by simple commands which can do a lot – i.e. by 'powerful' commands. In approximate order from low to high a selction of computer languages is listed here: machine code, assembly language, FORTRAN, BASIC, ALGOL, PASCAL, COBOL, LISP. Some languages such as FORTH, whilst of quite high level, have many features of both low and high level languages and it is difficult to say exactly how high a level they are.

MACHINE CODE

This is the actual numbers that the Z80 microprocessor chip uses. Not merely data, but every instruction which the chip executes is represented as far as the chip is concerned by machine–code numbers, and these, controlling internal voltages, are all it can use.

MEMORY LOCATION

The computer has lots of chips in it, in which to remember data. It stores up data in little groups of 8 bits, called 'bytes'. The computer has lots of these bytes in its memory; we call each of the places that holds a byte a 'memory location'. Each memory location has a special number associated with it, just like a house number: this number is called the 'address' of that memory location. It helps the computer find the byte it wants.

MEMORY MAP

A chart showing how the memory is used by the computer, i.e. which memory locations are used for what.

MNEMONICS

Pronounced 'nemoniks', and named after the Greek goddess of memory, Mnemosyne, mnemonics are simply memory aids. It is easier to understand or remember what 'ADD A,B' does than to remember what the Z80 chip will do with the instruction '80'. Hence we tend to use assembly language with its mnemonics rather than enter programs directly in machine code.

NESTING

A structure in a computer program where one loop is put inside another.

NYBBLE

A nybble is a chunk of 4 bits just as a byte is a chunk of 8 bits. For convenience a byte is sometimes considered to consist of two nybbles, the least significant nybble (right–hand one) and the most significant nybble (left–hand one).

OBJECT CODE

The machine code version of an assembly language program. The assembler is said to convert 'source code' into 'object code'.

OBJECT PROGRAM

The object program is the actual machine–code version of an assembly–language program (the source program).

OP CODE

An instruction, usually written in assembly language or in a higher level language, which the computer can follow: eg. ADD, LD, PRINT, etc. Note that the op code is the instruction minus its operands. Thus ADD A,B is an instruction, whilst ADD is the op code.

OPERATING SYSTEM

A built–in program which is designed to simplify 'housekeeping' procedures within the computer. It deals with such functions as scanning the keyboard, creating the video display, saving programs to tape, etc.

OPERAND

The operand of an instruction is any item which follows the command itself, such as A and 145 in LD A,145. Some instructions don't need an operand such as RET.

PARAMETER

Most instructions have parameters. For example in the BASIC line 'PRINT a,b'; a and b are the parameters.

PERIPHERAL

A device attached to the main computer system, such as a printer, VDU or light pen.

PIXEL

Short for 'picture element', a pixel is the smallest portion of the screen which the computer can control – i.e. a dot. All pictures, letters etc. which the computer puts on the screen are made up from combinations of pixels.

PROGRAM COUNTER

A special 16 bit register, in the Z80, that points to the next instruction to be executed, so that the Z80 does not lose its place in a program!

RAM

Abbreviation for **R**ead **A**nd **M**odify memory, also sometimes known as **R**andom **A**ccess **M**emory.

REGISTER

A special memory location in a chip, usually in the Central Processor Unit, (CPU) for storing a number, in binary. The registers in the Z80 are either 8 or 16 bits long.

RESET

A flag or a bit is 'reset' when it has the value '0'. A pixel is reset when it is not lit up.

ROM

Abbreviation for **R**ead **O**nly **M**emory. This is memory in which there is a program permantly stored, and cannot be erased by trying to POKE data into it or by means of LD type instructions.

SCREEN (Addresses)

The screen has a special area of the computer's memory reserved for it. This memory area tells the system what information goes where on the screen. If we alter the contents of these locations then the displayed character on the screen will change.

SERVICE ROUTINE

Machine–code routine used for servicing interrupts.

SET

Generally, 'set' is used to refer to flags; it means to put a 1 into the flag. A pixel is set when it is lit up. See also RESET.

SOURCE CODE

Assembly language as typed in, before converion to machine code or 'object code' by the assembler.

SOURCE PROGRAM

The source program consists of the actual assembly mnemonics, as used in this book. It means a program that was created by the user but must first be translated into some other form before execution can take place. An assembly language program must be changed into machine code in order to execute it.

SYMBOLIC LABEL

This is a name given to a line in a program, so that it can be referred to easily by the programmer. Instead of jumping to a specific memory location within a program, it can jump to the name instead.

VECTOR ADDRESS

The address that must be 'looked up' in an indirection operation is called the vector address.

Z80

Z80 is the type or reference number of the central processing unit (CPU) used in this computer. The CPU is often referred to as 'The Z80'.

Index

A

accessing screen 1–3
accumulator 1–2
ADC A,s 4–6
ADD A,n 1–2, 4–1
ADD A,s 4–6
ADD IX,pp 7–10
ADD IY,rr 7–10
adding 1–2, 4–1
address 1–2
  calculating jumps 2–4
addressing 1–2
  direct 3–2
  immediate 1–2, 1–9
  implied 1–10
  indexed 3–13
  indirect 3–9
  inherent 1–10
  register–register 1–9
alternate registers 9–5
AND A 4–5
AND s 5–4
architecture 1–8
ASCII 1–7, A7
assembly language 1–1

B

BCD 5–1, A5
binary
  coded decimal 5–1, A5
  division 6–9
  multiplication 6–2
  numbers A5
bit 2–8
BIT b,r 6–14
block moves 8–1
borrow 4–8
byte 2–4

C

calculating jump addresses 2–4
CALL cc,nn 2–19
CALL nn 2–18
carry flag 2–17, 4–6, 4–14
  reset 4–5
CCF 4–14
comments A6–11
comparisons 8–5
conditional jumps 2–8, 2–22, 8–3
CPD 8–9
CPDR 8–9
CPI 8–7
CPIR 8–8
CPL 5–14
CP s 8–6
crash 1–11
CURSET routine 7–5

D

DAA 5–2
DEC d 2–8
decrementing 2–8
DEFB 158 1–3, 1–12

DEFB n A6–9
DEFM "s" A6–9
DEFS n A6–7
DEFW nn A6–9
DI 9–2
direct addressing 3–2
direct–entry command 1–4
directive 1–4
disabling interrupts 9–2
division 6–9
DJNZ e 6–9
double precision 4–5

E

EI 9–2
enabling interrupts 9–2
ending a program 1–4, 1–5
ENT A6–7
exclusive or 5–9
EX AF,AF' 9–6
ED DE,HL 9–5
EQU nn 10–2, A6–8
EX (SP),HL 7–9, 9–7
EX (SP),IX 7–9, 9–7
EX (SP),IY 7–9, 9–7
EXX 9–5

F

fill routines 10–5
flags 2–8
    carry (C) 2–17, 4–6, 4–14
    half–carry (H) 2–18
    parity/overflow (P/V) 2–17
    sign (S) 2–18
    subtract (N) 2–17, 4–8
    zero (Z) 2–8, 2–18
flags register 2–8

G

graphics 3–4, 10–1

H

half–carry flag 2–18
HALT 9–7
hexadecimal A5

I

IM 0 9–3
IM 1 9–4
IM 2 9–5
immediate addressing
    of memory 1–2
    of registers 1–9
implied addressing 1–10
IN A,(n) 9–8
IN r,(C) 9–8
INC d 2–10
incrementing 2–10
IND 9–9
INDR 9–9
indexed addressing 3–13
index registers 3–4, 3–13
indirect addressing 3–9
inherent addressing 1–10
INI 9–9
INIR 9–9
interrupts 9–1
    modes 9–3

J

JP nn 2–1
JR e 2–5
JR NZ,e 2–9
jumps 2–1
    calculating addresses 2–4
    conditional 2–8, 8–3
    relative 2–5
    unconditional 2–1

L

labels 2–10
LD A,(BC) 3–9
LD A,(DE) 3–9
LD A,n 1–2
LD (BC),A 3–10
LDD 8–6
LD dd,nn 3–1
LD dd,(nn) 3–2
LD (DE),A 3–10
LDDR 8–6
LD (HL),r 3–9
LDI 8–1
LDIR 8–4
LD (nn),dd 3–2, 7–7
LD r,(HL) 3–9
LD r,n 1–9
LD r1,r2 1–10
LD SP,HL 7–7
LD SP,IX 7–7
LD SP,IY 7–7
LSB 3–7

M

machine code 1–1
machine calls 1–3, A4
MCALS 1–3, A4
memory 3–13
microprocessor 1–1
minus (N) flag 2–17
mnemonics 1–2
MSB 3–7
multiplicand 6–1
multiplication 6–1
multiplier 6–1

N

N (subtract/minus/negative) flag 2–17
NEG 5–14
negative flag 2–17
NOP 9–11
nybble 5–1

O

O command 1–3
one's complement 5–10
operand 2–9
operator 2–9
ORG nn 1–4, A6–9
OR s 5–8
OTDR 9–10
OTIR 9–10
OUT (C),r 9–10
OUT (n),A 9–9
OUTD 9–10
OUTI 9–10
overflow (P/V) flag 2–17

P

parity 8–3
(P/V) flag 2–17
pointer, stack 7–6
POP IX 7–4
POP IY 7–4
POP qq 7–3
printing on screen 1–3
program counter 2–5
PUSH IX 7–3
PUSH IY 7–3
PUSH qq 7–3

R

register 1–2, 1–8
accumulator 1–2
addressing 1–8, 1–9, 1–10
alternate 9–5
exchanging 7–8
flags 2–8
pairs 3–1
index 3–4, 3–13
register–register addressing 1–9
relative jumps 2–5
RES b,r 6–15

reset (0) 2–8
    carry flag 4–5
restart instruction 1–3
RET 1–4, 2–18
RETI 9–2
RETN 9–2
RLCA 6–12
RL s 6–7
RRCA 6–12
rotating bits 6–7, 6–12
RST 8 1–3, 1–12
RST n 9–3

S

SBC A,s 4–8
SCF 4–14
scratchpad 3–4
screen access 1–3
set (1) 2–8
SET b,r 6–15
shifting bits 6–5
signed numbers 5–10
sign flag 2–18
SRL s 6–5
SP 7–6
sprites 10–8
stack 7–1
    pointer (SP) 7–6
subroutines 2–18
SUB s 4–8
subtract flag 2–17, 4–8
subtraction 4–8
symbolic labels 2–10
symbolic operations 3–15

T

two's complement 5–11

U

unconditional jumps 2–1

X

XOR 5–9

Z

Z80 chip 1–1, 1–8
zero flag 2–8, 2–17